Software Unit Testing Framework Project

By Krithiga Venkataraman

OBJECTIVES:

To apply unit-level testing to develop test for a Heap Algorithm and to develop unit-level test case using a generative AI tool and unit frame work and to analyze the effectiveness of generative AI too in performing unit-level testing.

Technology used:

Generative AI tool(ChatGPT)

Pytest-Unit Testing framework

Pycharm – IDE

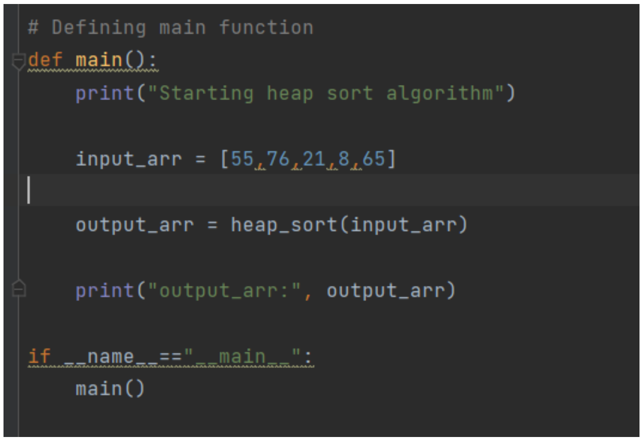
Project Description:

The Project to focus on deriving test case for a Heap Algorithm using pytest – unittesting framework and to developed test case using a pytest-unit testing framework

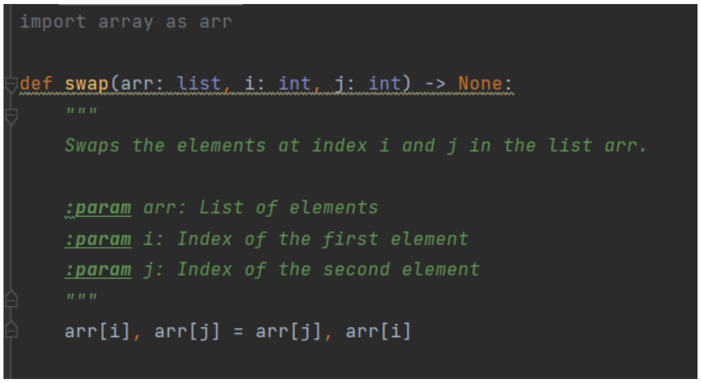
Task 1:

Development of the Heap Sort code.

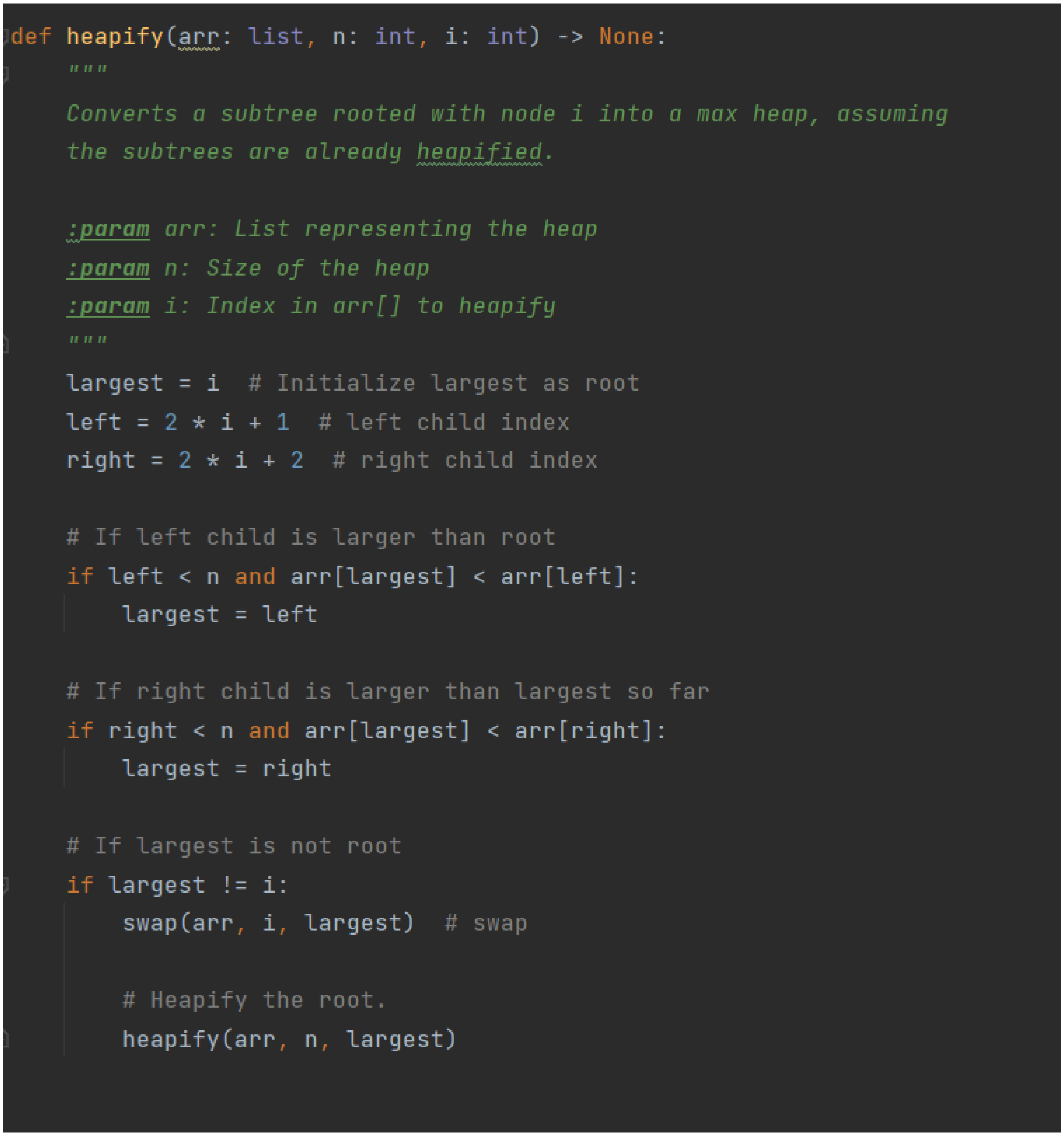
The below algorithm was downloaded from the internet.



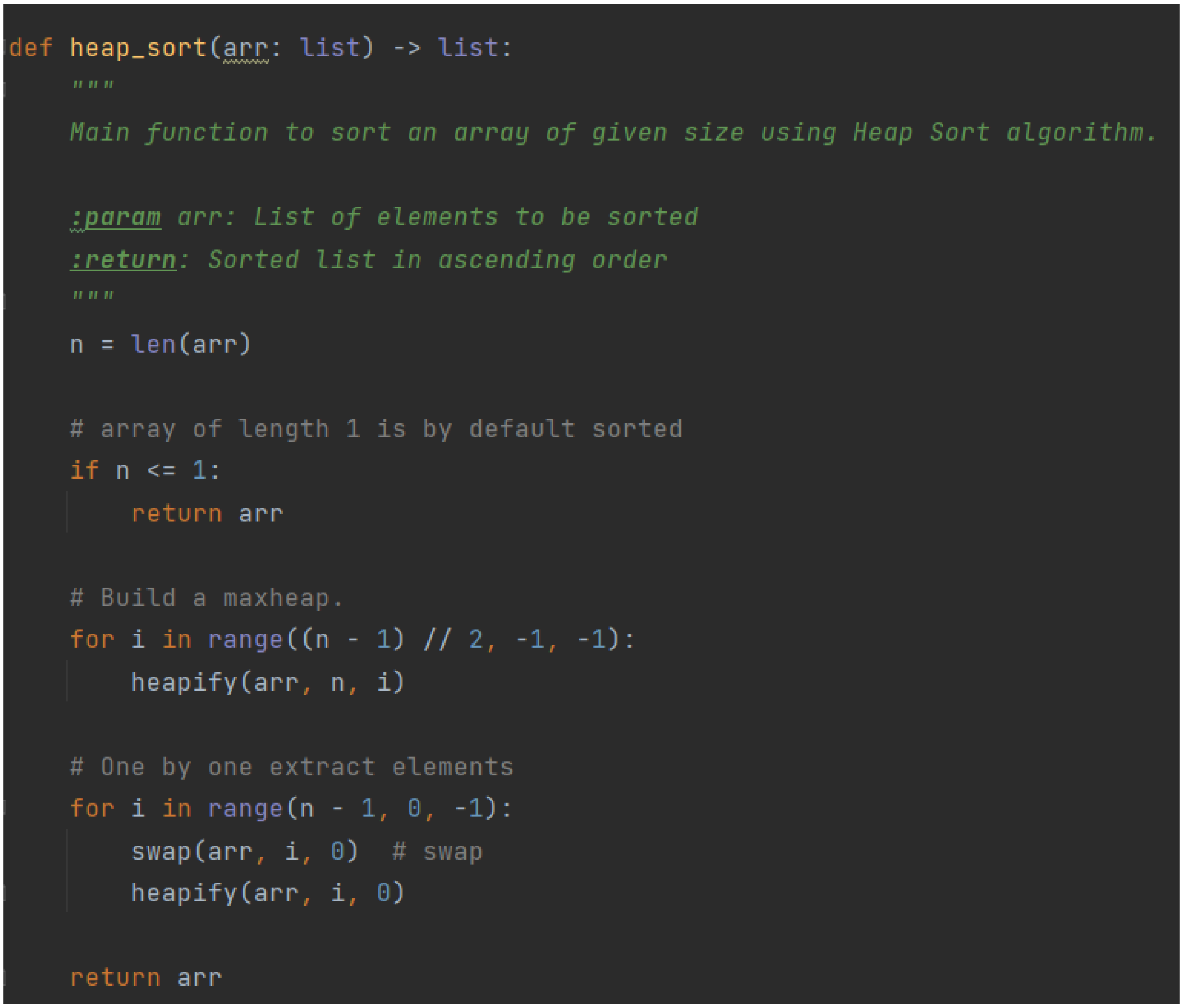
```
# Defining main function
def main():
    print("Starting heap sort algorithm")

    input_arr = [55,76,21,8,65]

    output_arr = heap_sort(input_arr)

    print("output_arr:", output_arr)

if __name__=="__main__":
    main()
```



```python
import array as arr

def swap(arr: list, i: int, j: int) -> None:
    """
    Swaps the elements at index i and j in the list arr.

    :param arr: List of elements
    :param i: Index of the first element
    :param j: Index of the second element
    """
    arr[i], arr[j] = arr[j], arr[i]
```

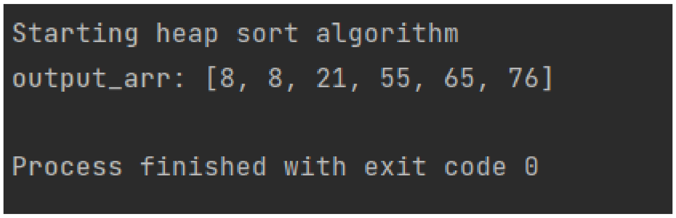


```python
def heapify(arr: list, n: int, i: int) -> None:
    """
    Converts a subtree rooted with node i into a max heap, assuming
    the subtrees are already heapified.

    :param arr: List representing the heap
    :param n: Size of the heap
    :param i: Index in arr[] to heapify
    """
    largest = i  # Initialize largest as root
    left = 2 * i + 1  # left child index
    right = 2 * i + 2  # right child index

    # If left child is larger than root
    if left < n and arr[largest] < arr[left]:
        largest = left

    # If right child is larger than largest so far
    if right < n and arr[largest] < arr[right]:
        largest = right

    # If largest is not root
    if largest != i:
        swap(arr, i, largest)  # swap

        # Heapify the root.
        heapify(arr, n, largest)
```



```python
def heap_sort(arr: list) -> list:
    """
    Main function to sort an array of given size using Heap Sort algorithm.

    :param arr: List of elements to be sorted
    :return: Sorted list in ascending order
    """
    n = len(arr)

    # array of length 1 is by default sorted
    if n <= 1:
        return arr

    # Build a maxheap.
    for i in range((n - 1) // 2, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n - 1, 0, -1):
        swap(arr, i, 0)  # swap
        heapify(arr, i, 0)

    return arr
```

Program Output:



```
Starting heap sort algorithm
output_arr: [8, 8, 21, 55, 65, 76]

Process finished with exit code 0
```

Algorithm:

```
function heapify(arr, n, i):
largest = i        // Initialize largest as root
 left = 2 * i + 1     // left = 2*i + 1
  right = 2 * i + 2    // right = 2*i + 2
// If left child is larger than root
```

```
  if left < n and arr[left] > arr[largest]:
     largest = left
 // If right child is larger than largest so far
  if right < n and arr[right] > arr[largest]:
     largest = right
 // If largest is not root
  if largest != i:
     swap(arr[i], arr[largest])
 // Recursively heapify the affected sub-tree
     heapify(arr, n, largest)
```

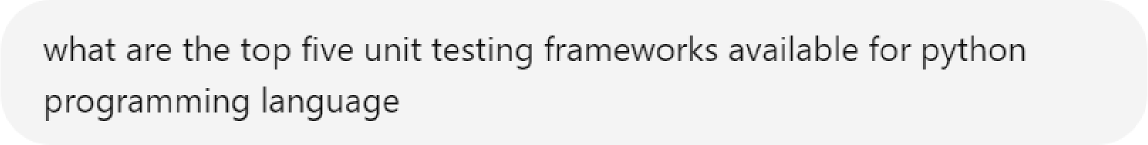
Unit Testing framework and prompt generation:

A unit testing framework is a software tool or library that provides a structured environment for writing, executing, and managing unit tests in software development. It helps simplify the process of writing, organizing and managing tests. It is also an integral and fundamental part of the software testing process, where individual components of the application are tested in isolation to ensure they function correctly (https://www.lambdatest.com/).

There are many unit testing frameworks available for the python programming language, each with its own set of unique features and performance.

For the GenAI tool, I wanted to identify one which was good at natural language understanding and good at being a coding assistant. I decided to go with ChatGPT which met both my criteria and was one of the first GenAI tools available to the general public and with the assumption that it should have better training models based on the real-world usage.

Here are few of the top unit testing frameworks available for python based on below prompt used in ChatGPT



what are the top five unit testing frameworks available for python programming language

ChatGPT response:

In Python, there are several excellent unit testing frameworks available. Here are five of the top ones:

1. **unittest**:

   - **Overview**: This is Python’s built-in unit testing framework. It’s inspired by Java’s JUnit and is included with the standard library.
   - **Features**: It provides test case creation, test discovery, fixtures, and test runners. It also supports test grouping and test skipping.
   - **Usage**: Ideal for users who prefer a built-in solution with no extra dependencies.

2. **pytest**:
   - **Overview**: Pytest is one of the most popular third-party testing frameworks. It is known for its simplicity and powerful features.
   - **Features**: It supports fixtures, parameterized testing, and has a rich ecosystem of plugins. Its syntax is more concise compared to unittest, and it provides detailed failure reports.
   - **Usage**: Great for both simple and complex testing scenarios, and it has extensive documentation and community support.
3. **nose2**:
   - **Overview**: Nose2 is the successor to the now-unmaintained Nose framework. It aims to extend unittest and provides a range of plugins to enhance testing.
   - **Features**: It includes test discovery, plugins, and test result reporting. It’s designed to be compatible with unittest test cases.
   - **Usage**: Suitable for users who need additional features beyond unittest and prefer a more flexible, plugin-oriented approach.
4. **doctest**:
   - **Overview**: This is another built-in framework that allows you to write tests as part of your documentation. It’s included in Python’s standard library.
   - **Features**: It extracts examples from docstrings and runs them as tests, verifying that they produce the expected results.
   - **Usage**: Best for simple, example-driven testing where you want to ensure that your documentation examples are accurate.
5. **hypothesis**:
   - **Overview**: Hypothesis is a property-based testing framework, which means it generates a wide range of input data to test your code.
   - **Features**: It integrates with pytest and unittest, and it’s particularly useful for identifying edge cases and unexpected behaviors by generating a variety of test cases.

    - **Usage**: Ideal for more rigorous testing where you want to explore a broad range of input scenarios.

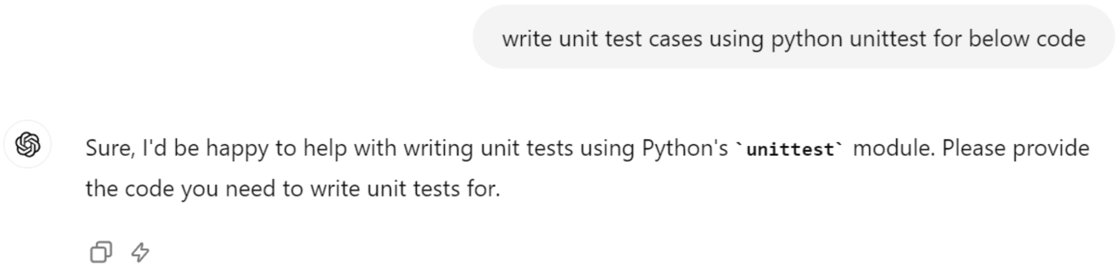
Based on the above features and usage descriptions, I decided to go with unittest, which is Python's built-in unit testing framework & which requires minimum dependencies. It is included with the standard Python library and has the features I was looking for.

Next, I proceeded to generate the unit test cases for the heapsort code by providing prompts to ChatGPT. Based on my analysis of the best practices in providing a prompt to any genAI tool, I wrote my prompt to have the following characteristics:

- Being specific
- Providing context
- Having action words

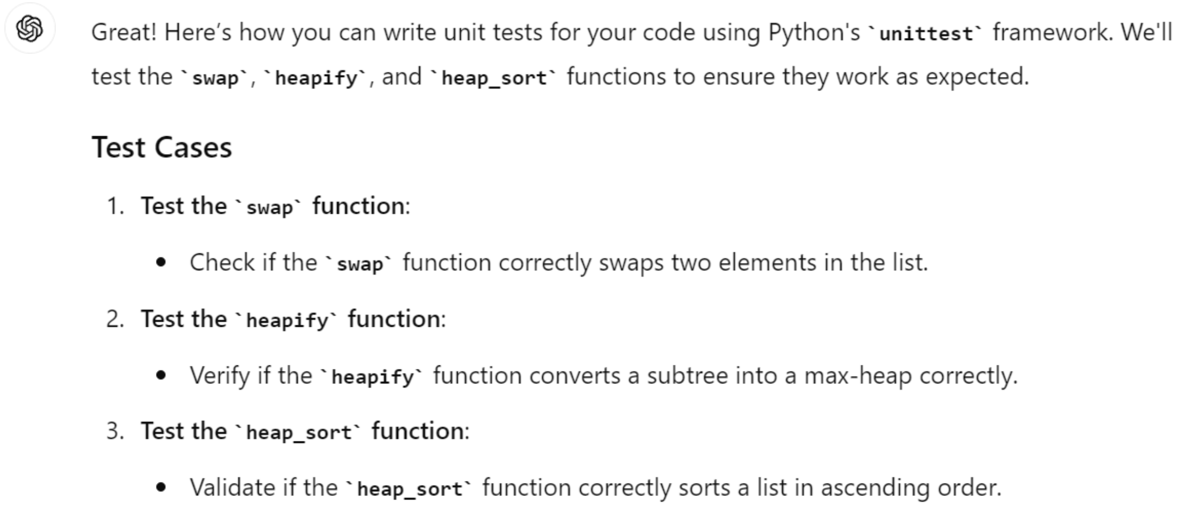
I experimented with few variations and ended with the following prompts.

Unit test cases generated by ChatGPT:



write unit test cases using python unittest for below code

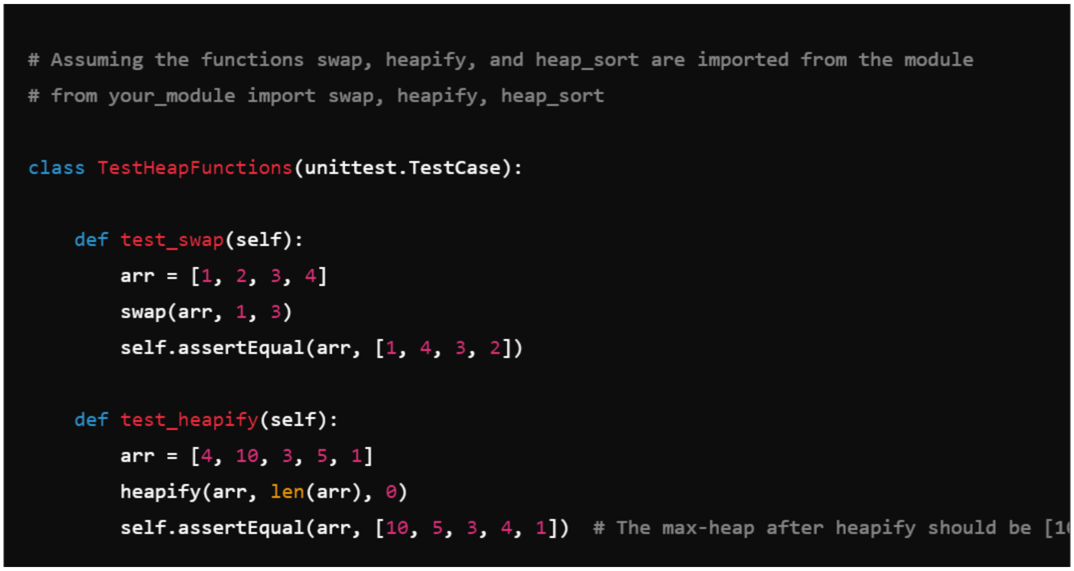
Sure, I'd be happy to help with writing unit tests using Python's `unittest` module. Please provide the code you need to write unit tests for.



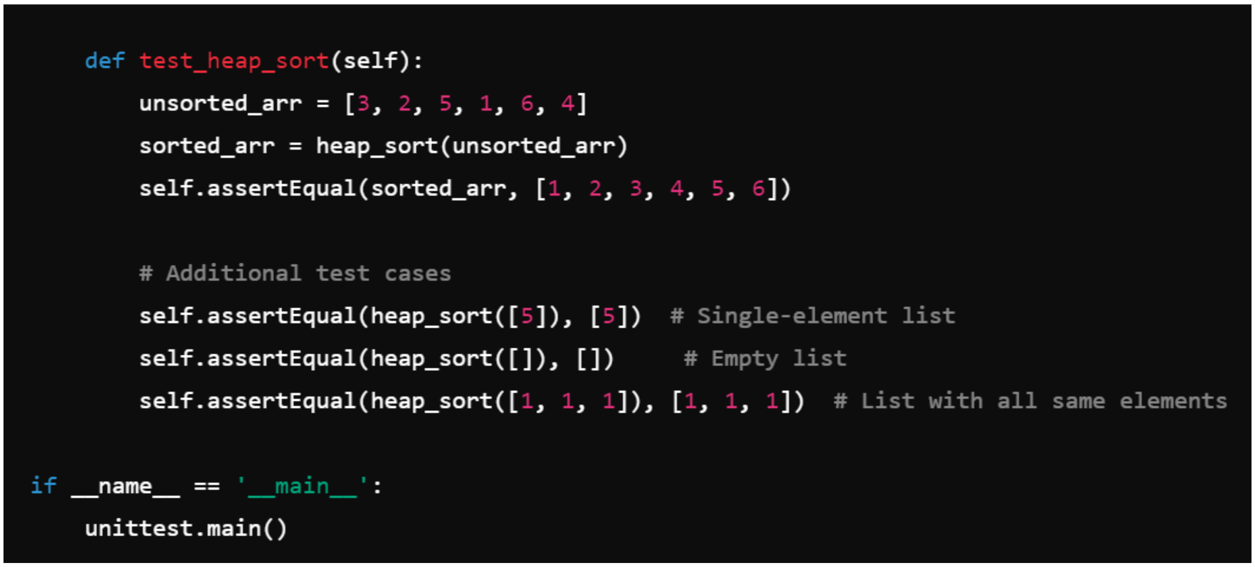
Great! Here's how you can write unit tests for your code using Python's `unittest` framework. We'll test the `swap`, `heapify`, and `heap_sort` functions to ensure they work as expected.

Test Cases

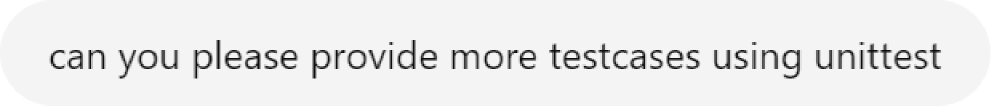
1. **Test the `swap` function**:
    - Check if the `swap` function correctly swaps two elements in the list.
2. **Test the `heapify` function**:
    - Verify if the `heapify` function converts a subtree into a max-heap correctly.
3. **Test the `heap_sort` function**:
    - Validate if the `heap_sort` function correctly sorts a list in ascending order.



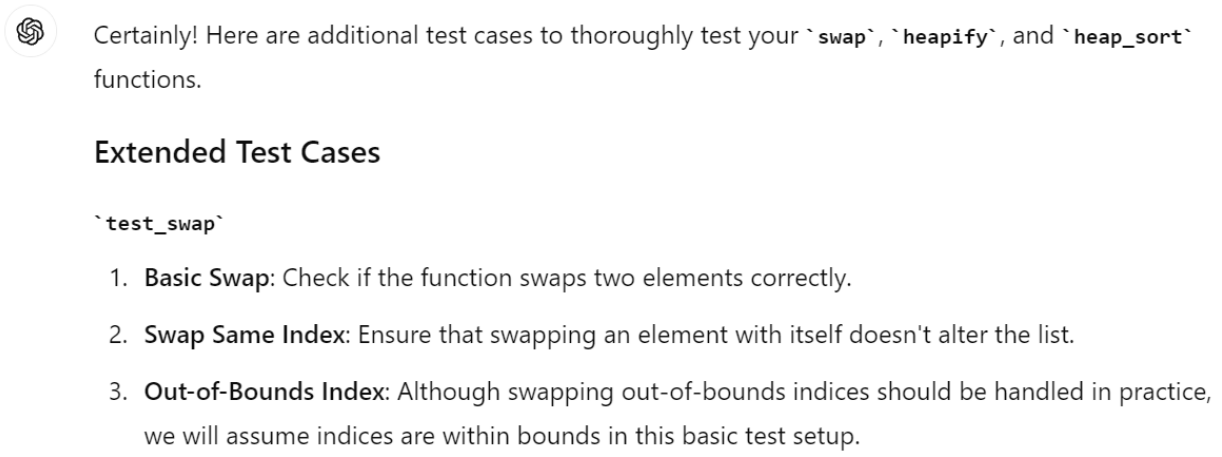
```
# Assuming the functions swap, heapify, and heap_sort are imported from the module
# from your_module import swap, heapify, heap_sort

class TestHeapFunctions(unittest.TestCase):

    def test_swap(self):
        arr = [1, 2, 3, 4]
        swap(arr, 1, 3)
        self.assertEqual(arr, [1, 4, 3, 2])

    def test_heapify(self):
        arr = [4, 10, 3, 5, 1]
        heapify(arr, len(arr), 0)
        self.assertEqual(arr, [10, 5, 3, 4, 1])  # The max-heap after heapify should be [1
```



```
    def test_heap_sort(self):
        unsorted_arr = [3, 2, 5, 1, 6, 4]
        sorted_arr = heap_sort(unsorted_arr)
        self.assertEqual(sorted_arr, [1, 2, 3, 4, 5, 6])

        # Additional test cases
        self.assertEqual(heap_sort([5]), [5])  # Single-element list
        self.assertEqual(heap_sort([]), [])     # Empty list
        self.assertEqual(heap_sort([1, 1, 1]), [1, 1, 1])  # List with all same elements

if __name__ == '__main__':
    unittest.main()
```



can you please provide more testcases using unittest

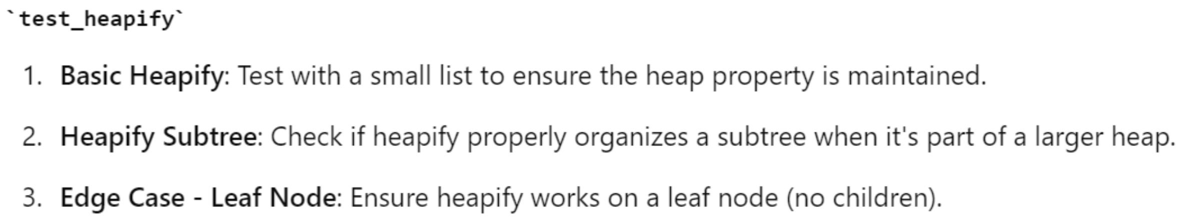


Certainly! Here are additional test cases to thoroughly test your `swap`, `heapify`, and `heap_sort` functions.

Extended Test Cases

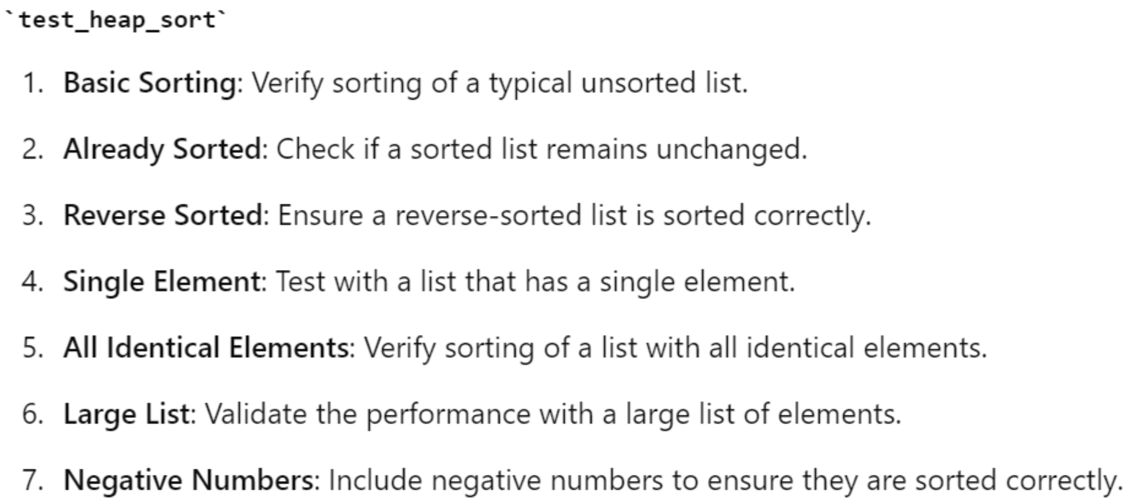
`test_swap`

1. **Basic Swap**: Check if the function swaps two elements correctly.
2. **Swap Same Index**: Ensure that swapping an element with itself doesn't alter the list.
3. **Out-of-Bounds Index**: Although swapping out-of-bounds indices should be handled in practice, we will assume indices are within bounds in this basic test setup.



`test_heapify`

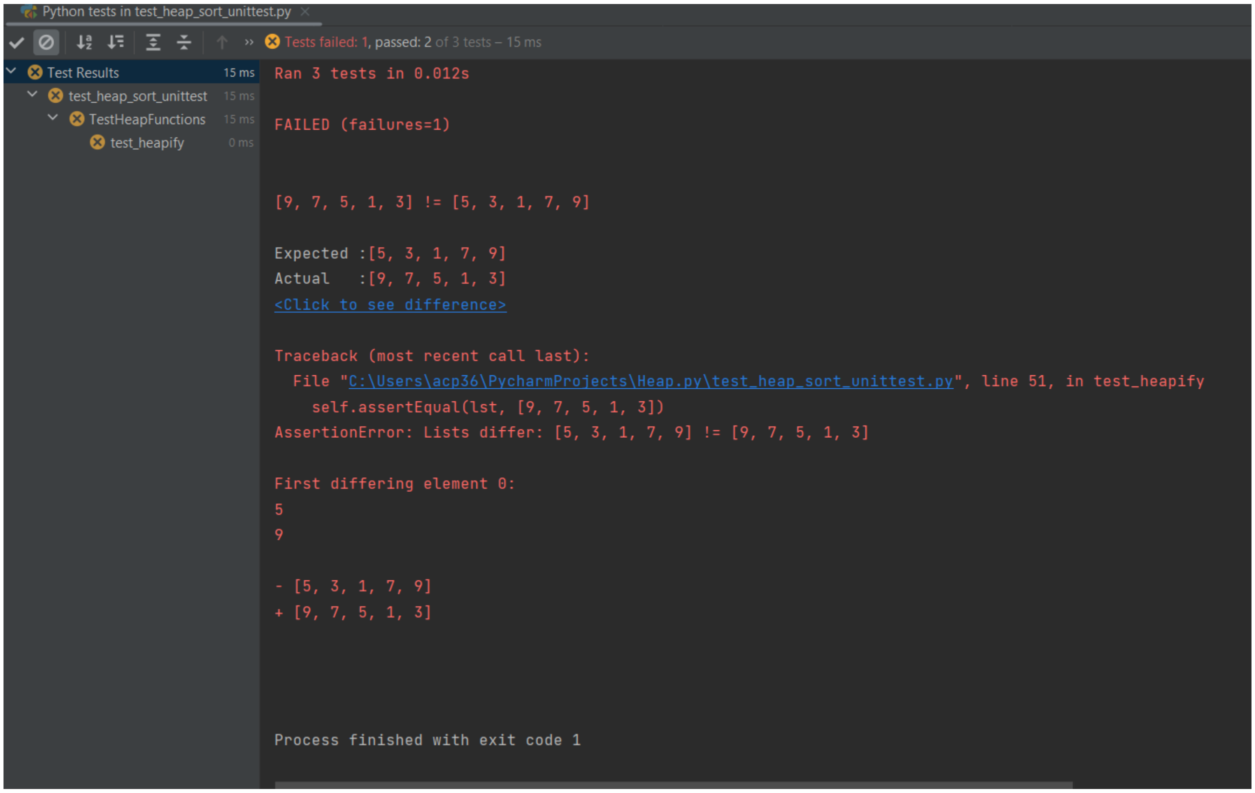
1. **Basic Heapify**: Test with a small list to ensure the heap property is maintained.
2. **Heapify Subtree**: Check if heapify properly organizes a subtree when it's part of a larger heap.
3. **Edge Case - Leaf Node**: Ensure heapify works on a leaf node (no children).



`test_heap_sort`

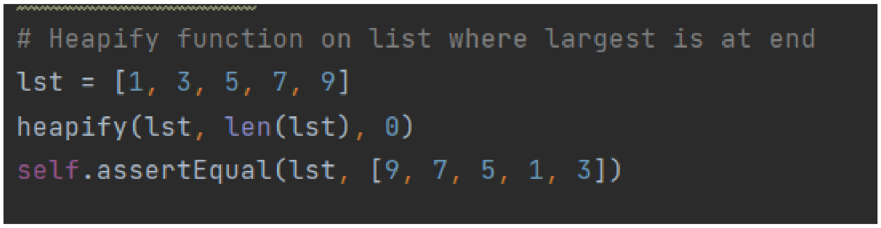
1. **Basic Sorting**: Verify sorting of a typical unsorted list.
2. **Already Sorted**: Check if a sorted list remains unchanged.
3. **Reverse Sorted**: Ensure a reverse-sorted list is sorted correctly.
4. **Single Element**: Test with a list that has a single element.
5. **All Identical Elements**: Verify sorting of a list with all identical elements.
6. **Large List**: Validate the performance with a large list of elements.
7. **Negative Numbers**: Include negative numbers to ensure they are sorted correctly.

Screenshot of the results:


Python tests in test_heap_sort_unittest.py
Tests failed: 1, passed: 2 of 3 tests – 15 ms
Test Results 15 ms
test_heap_sort_unittest 15 ms
TestHeapFunctions 15 ms
test_heapify 0 ms

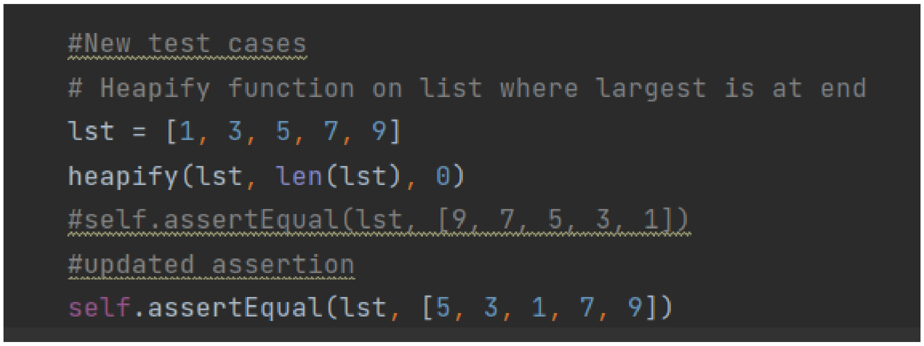
```
Ran 3 tests in 0.012s

FAILED (failures=1)


[9, 7, 5, 1, 3] != [5, 3, 1, 7, 9]

Expected :[5, 3, 1, 7, 9]
Actual   :[9, 7, 5, 1, 3]
<Click to see difference>

Traceback (most recent call last):
  File "C:\Users\acp36\PycharmProjects\Heap.py\test_heap_sort_unittest.py", line 51, in test_heapify
    self.assertEqual(lst, [9, 7, 5, 1, 3])
AssertionError: Lists differ: [5, 3, 1, 7, 9] != [9, 7, 5, 1, 3]

First differing element 0:
5
9

- [5, 3, 1, 7, 9]
+ [9, 7, 5, 1, 3]



Process finished with exit code 1
```

As we can see from the above unit test results screenshot, one of the tests failed and the others ran successfully. This test case on the heapify function failed because the expected output provided to the assert function was incorrectly generated by the AI tool. This was again surprising that it can make such basis errors and requires user intervention to make them work.

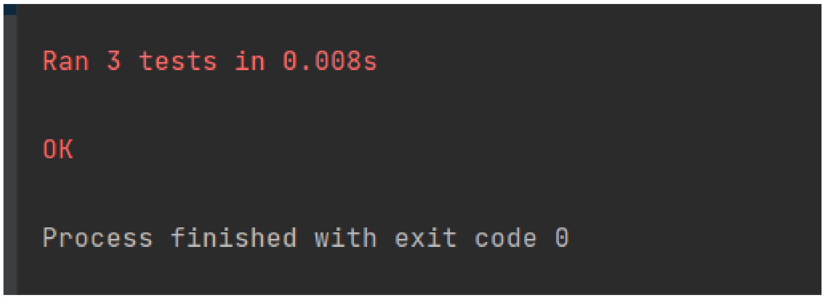


```
# Heapify function on list where largest is at end
lst = [1, 3, 5, 7, 9]
heapify(lst, len(lst), 0)
self.assertEqual(lst, [9, 7, 5, 1, 3])
```

I went ahead and fixed the assertion output and then the test passed.



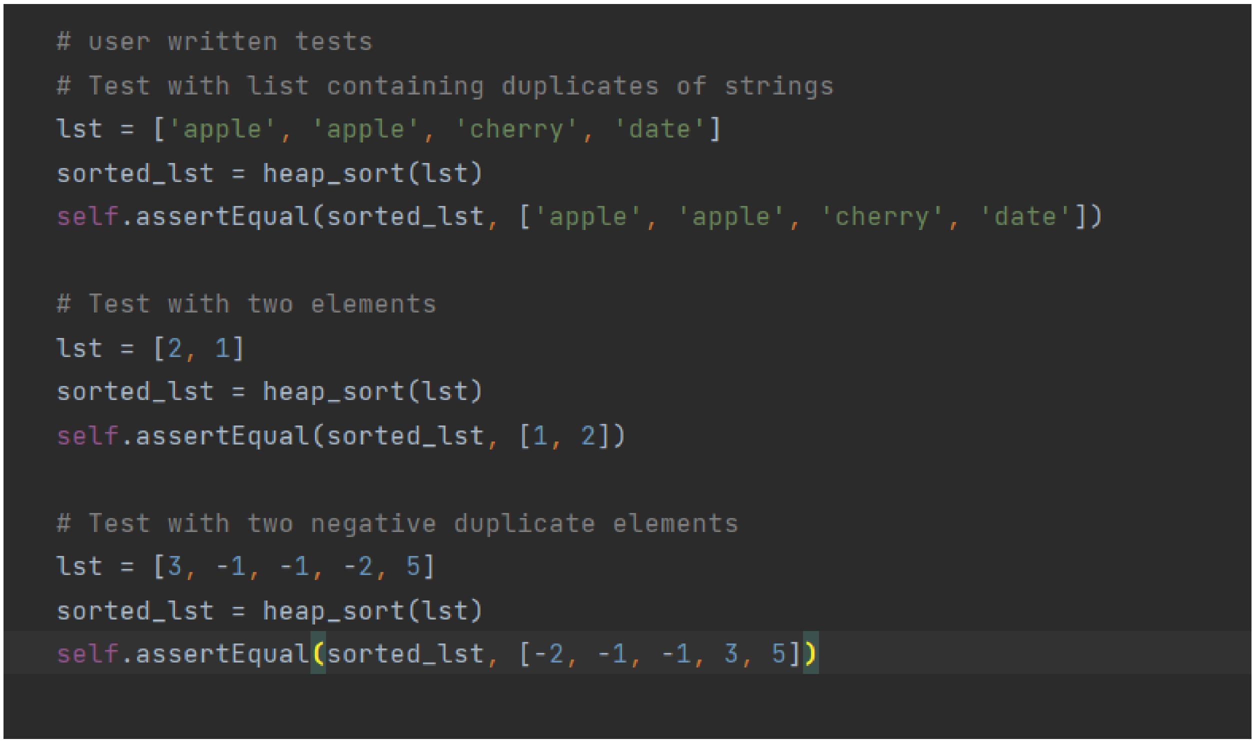
```
    #New test cases
    # Heapify function on list where largest is at end
    lst = [1, 3, 5, 7, 9]
    heapify(lst, len(lst), 0)
    #self.assertEqual(lst, [9, 7, 5, 3, 1])
    #updated assertion
    self.assertEqual(lst, [5, 3, 1, 7, 9])
```



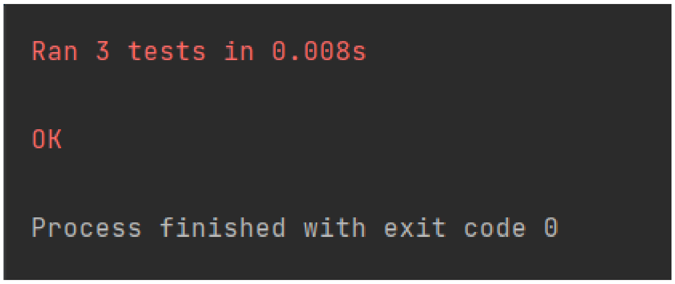
```
Ran 3 tests in 0.008s

OK

Process finished with exit code 0
```

Developer supplied test cases

I looked at the coverage of test cases and decided to supplement it with the following additional test cases.



```python
# user written tests
# Test with list containing duplicates of strings
lst = ['apple', 'apple', 'cherry', 'date']
sorted_lst = heap_sort(lst)
self.assertEqual(sorted_lst, ['apple', 'apple', 'cherry', 'date'])

# Test with two elements
lst = [2, 1]
sorted_lst = heap_sort(lst)
self.assertEqual(sorted_lst, [1, 2])

# Test with two negative duplicate elements
lst = [3, -1, -1, -2, 5]
sorted_lst = heap_sort(lst)
self.assertEqual(sorted_lst, [-2, -1, -1, 3, 5])
```



```
Ran 3 tests in 0.008s

OK

Process finished with exit code 0
```

Assessment of ChatGPT test cases

The initial test cases provided by the AI tool were basic and did not include all the boundary and edge cases. It was however impressive that it was able to generate code in the language and the framework mentioned in the prompt. It was able to generate complete classes with the required imports to run the test cases. I had to make minimum changes and just change the module which had the original code to be tested to run the code.

However, the first set of test cases provided were not sufficient to get 100% code coverage nor build the confidence that the heapsort program would work under all circumstances.

Also, I noticed that the same prompt when run at different times or in quick succession, yielded different results. Sometimes, the test cases had comments on them and sometimes they did not. The number of test cases also varied each time with the same prompt.

I then was curious if the GenAI tool would be able to provide me with more test cases, if I prompted for it and to my surprise it did. With the prompt, “can you please provide me more test cases with unittest”, it provided additional test cases covering more boundary conditions. Also, it understood that the context that I am asking for the test cases on the program I had already supplied in the initial prompt.

In conclusion, I think GenAI has come a long way, and I find it to be a great assistant to developers & testers in helping write test cases. It did help me with good test case recommendations and helped save lot of time in coming up with these tests. However, at the same time, it is not 100% reliable and would need human intervention to verify and approve the tests written by a AI tool.

Reference:

Makas.S. (2023, July 26). Heap in python. Site Medium.

https://medium.com/@sarperismetmakas/heaps-in-python