Bonus task: `client_test.py`

- Unit tests are useful in software development:
  - It helps increase developers' confidence in the code they write
  - It helps us to design better robust / maintainable / reusable code
  - It helps us document how our code works for other developers to use.
  - It helps us catch errors in code modifications early on

- Unit tests have nothing to do with having a running client application. They run independently from the main application and exist for the reasons stated above. The test cases we feed into the methods we test for represent the use cases we anticipate for those methods

- In the case of this task, the methods we want to test are **getDataPoint** and **getRatio.** We want to cover these with tests so that we ensure these methods are returning the correct output as individual methods thereby ensuring us that when we use them together in the main client application, things will work fine.

- In unit testing, we always follow the pattern, Build-Act-Assert:
  - Arrange: We first arrange a simulated test scenario e.g. instantiating the dummy data we will pass in the methods we'll test, importing the class whose methods we want to test, etc.
  - Act: We then make some operations and call the method we want to test for
  - Assert: We check if the output of the method we're testing matches the expectation we have (e.g. dummy / static data of the outcome)

- Unit tests mostly use dummy data to represent the test cases we pass into methods and the expected outcome of these methods when such test cases are given. This is fine because we want our methods and tests to be deterministic and not randomly failing / behaving in a different way we don't expect.

- For this part of the task, you atleast have to add assertions to the test methods that already exist in the `client_test.py` file.

- An assertion would look something like the following bullet points below:
  - `self.assertEqual(getDataPoint(quote),dataPoint)`
  - `self.assertEqual(1,1)`

- For more examples on assertions in python unit testing check this resource or other online resource about Python unit testing

- As a rule of thumb, aim to use only one assertion. Use the strongest assertion you can think of. For instance, in this case, having multiple asserts to the output of **getDataPoint** isn't advisable. Instead, since the **getDataPoint** returns a tuple, assert that it returns a tuple containing the data you expect.

- If you want to take things a step further, consider adding more tests. Only **getDataPoint** is covered so far, so you can try to cover for **getRatio** as well e.g. when you have to compute for the ratio and you have the case where price B is zero, or when price A is zero, etc.

More Tips:

- To help get you started, for the initial tests that exist in the client_test.py file but are half complete, all you have to do is to add assertions to them, nothing more.

- The quotes is a list of individual quotes that you can pass into the **getDataPoint** function. Each quote represents a possible variation of quote data but if getDataPoint was modified properly, then it should return a tuple that contains (stock, top_bid_price, top_ask_price, stock_price_based_on_formula).. That's what you will have to assert for for each quote you pass in. (meaning you'll have to iterate quotes list and make assertions per iteration)

- Check the image on the next slide to preview the end result

```
def test_getDataPoint_calculatePrice(self):
    quotes = [
        {'top_ask': {'price': 121.2, 'size': 36}, 'timestamp': '2019-02-11 22:06:30.572453', 'top_bid': {'price': 120.48, 'size': 109}, 'id': '0.109974697771', 'stock': 'ABC'},
        {'top_ask': {'price': 121.68, 'size': 4}, 'timestamp': '2019-02-11 22:06:30.572453', 'top_bid': {'price': 117.87, 'size': 81}, 'id': '0.109974697771', 'stock': 'DEF'}
    ]
    """ ------------ Add the assertion below ------------ """
    for quote in quotes:
        self.assertEqual(getDataPoint(quote), (quote['stock'], quote['top_bid']['price'], quote['top_ask']['price'], (quote['top_bid']['price'] + quote['top_ask']['price'])/2))

def test_getDataPoint_calculatePriceBidGreaterThanAsk(self):
    quotes = [
        {'top_ask': {'price': 119.2, 'size': 36}, 'timestamp': '2019-02-11 22:06:30.572453', 'top_bid': {'price': 120.48, 'size': 109}, 'id': '0.109974697771', 'stock': 'ABC'},
        {'top_ask': {'price': 121.68, 'size': 4}, 'timestamp': '2019-02-11 22:06:30.572453', 'top_bid': {'price': 117.87, 'size': 81}, 'id': '0.109974697771', 'stock': 'DEF'}
    ]
    """ ------------ Add the assertion below ------------ """
    for quote in quotes:
        self.assertEqual(getDataPoint(quote), (quote['stock'], quote['top_bid']['price'], quote['top_ask']['price'], (quote['top_bid']['price'] + quote['top_ask']['price'])/2))
```

- Notice how the two tests have the same assertion. So what's the difference?

- The only difference is that the quotes list in the second test always contains quotes whose top_bid_price is greater than the top_ask_price. Hence the name of the test calculatePriceBidGreaterThanAsk, (*where bid_price is greater than ask_price*)