Testing Documentation

Cryptocurrency Price Predictor

Vincent Achukwu (17393546) & Joseph Lyons (16485216)

Supervisor: Dr. Martin Crane

Submission Date: 24/04/2022

Regression and Functional Testing

Throughout the development of this project, we tested the application ourselves by ensuring all features were working properly as expected and that there were no issues when testing different scenarios. At times, we would alter the code to test how certain features would work, such as how to implement the graphs for the market sentiment page. Instead of having an image displaying a bar chart as to whether the user-selected cryptocurrency has a positive or negative sentiment, we made us of an interactive piechart that describes the proportion of tweets that are positive or negative. After adding features to the application, we would ensure that the other features work as expected and that the addition of new features wouldn’t break the application.

Front End Visualisation Testing

Because our app has alot of visualisation features within it, Front end visualisation was a key part when designing the front end. We relied alot on manual testing to make sure that all features are displaying as expected.

Manual testing:

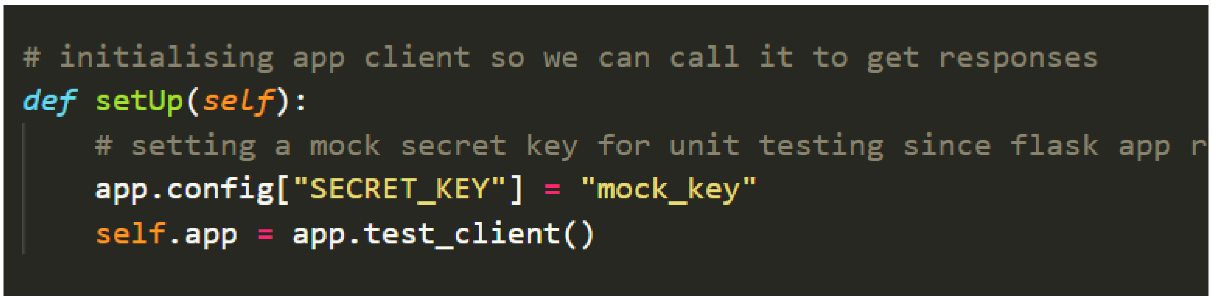
Manual testing was carried out every time new code was added to ensure nothing broke visually and everything was displayed correctly. With regular use of this made errors easier to fix as we wouldnt move on until displaying as we wanted. This was critical when displaying the data in our graphs as the twitter api was initaly collecting more than 100 tweets which we were able to fix. Also allowed us to choice a colour scheme for our pie chart that was easily visible to all users.

User Testing:

We carried out user testing ourselves to ensure the application was running correctly. We each done various walkthroughs of the app picking different variables to ensure consistancy was there each time.

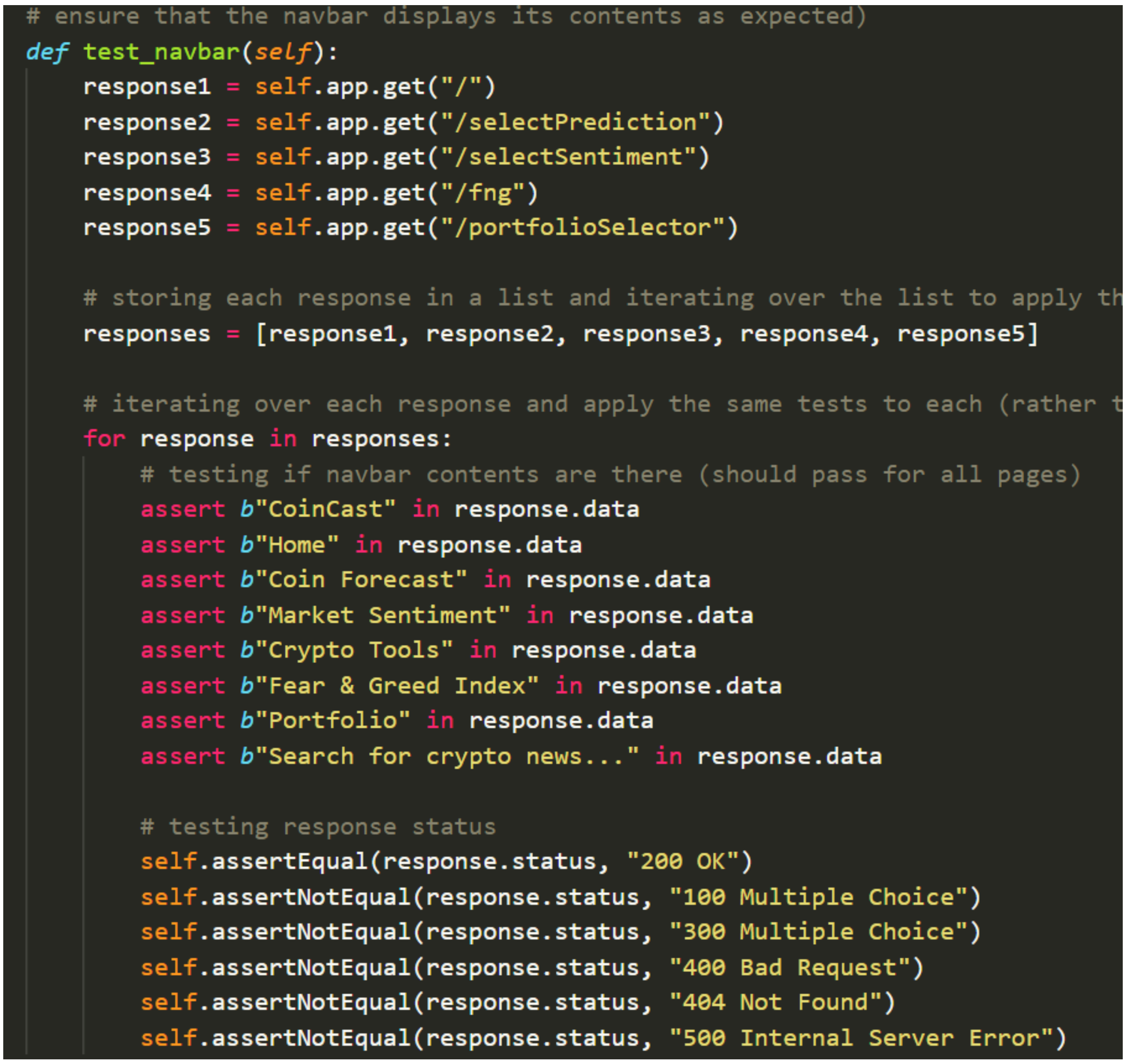
Unit Testing

For unit testing, we used the unittest library in Python. The main idea behind the unit tests for the backend was to test if each route had a response and if any route required some sort of input, we would hardcode any valid and invalid inputs and test if the response would be as expected. First, the application client is initialized and a temporary session key is created.



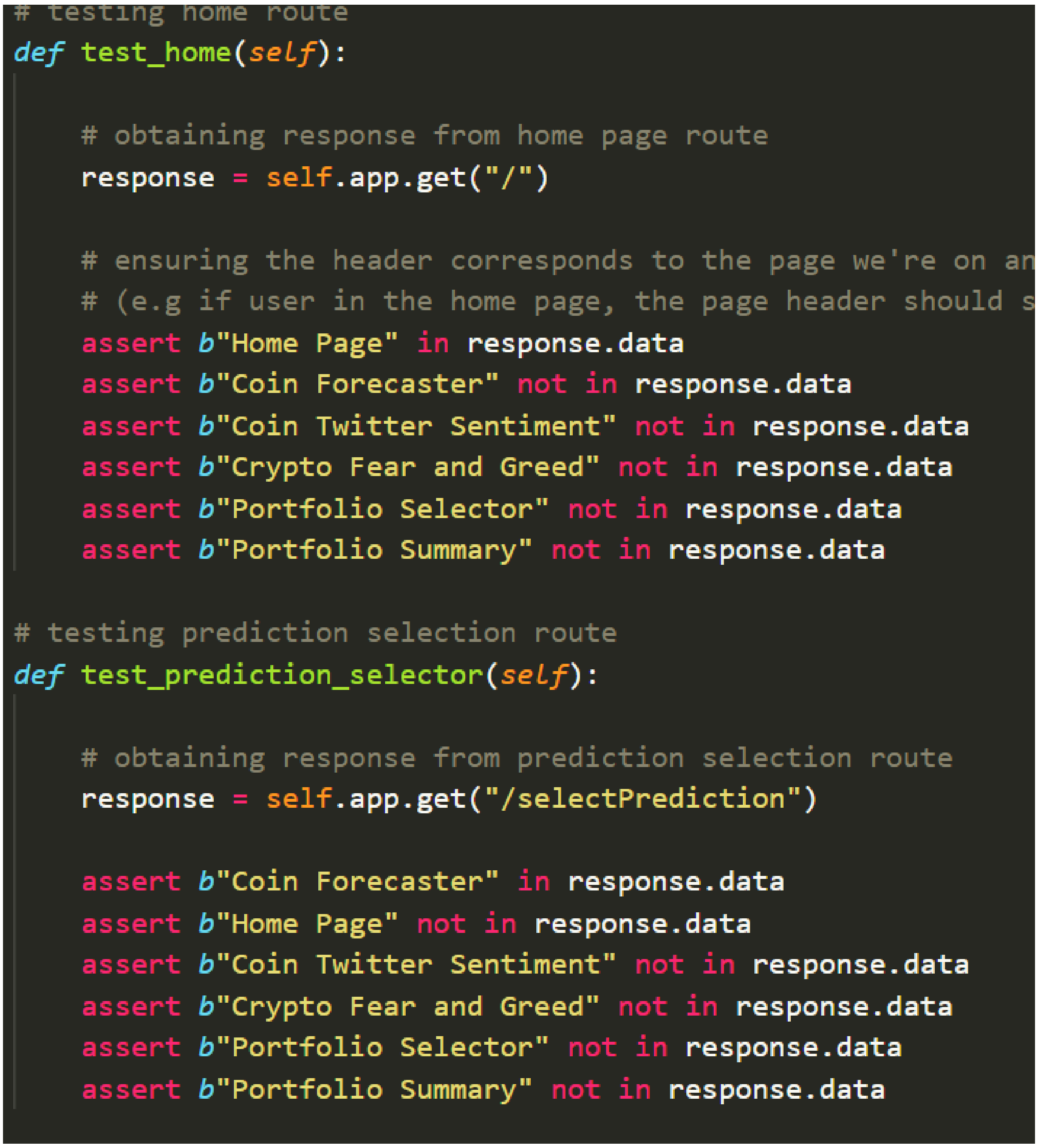
```
# initialising app client so we can call it to get responses
def setUp(self):
    # setting a mock secret key for unit testing since flask app r
    app.config["SECRET_KEY"] = "mock_key"
    self.app = app.test_client()
```

Then the routes are separated into different responses after which they are appended to a list that is iterated over and the same tests are applied to each response. The purpose of this approach (which can be found throughout the unit test code) is to avoid the repetition of the same type of tests for different routes. Here, the contents of the navbar are being tested for each of the routes which can be accessed by the user. The response status is also being tested for each page.

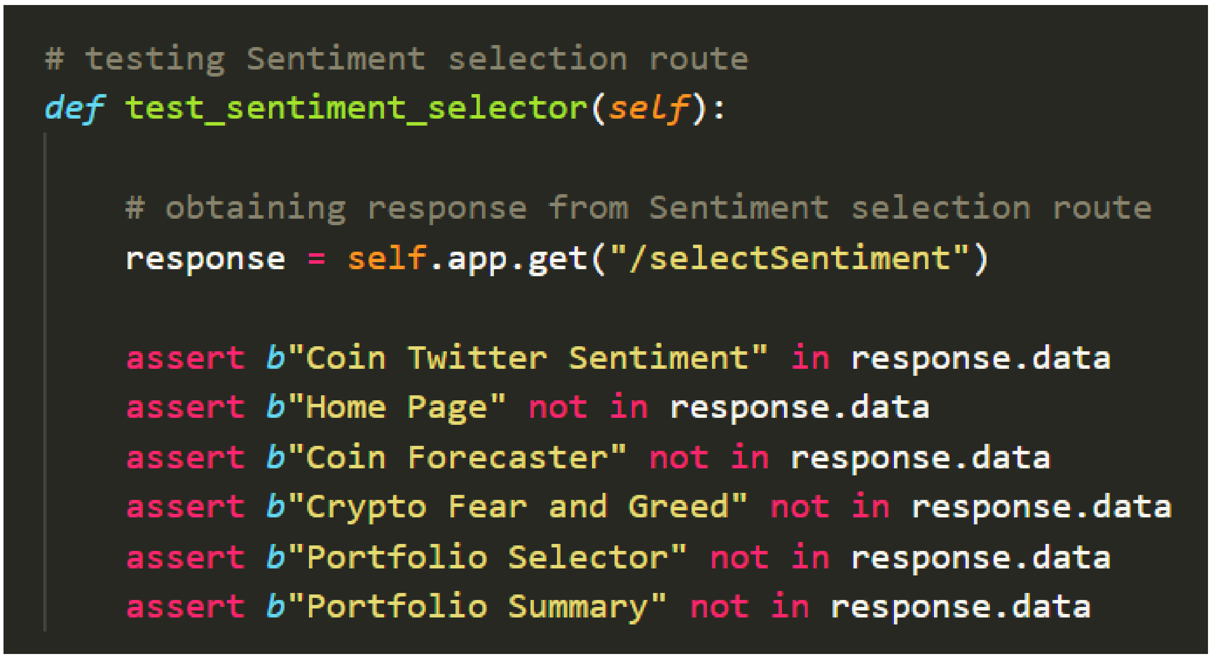


```
# ensure that the navbar displays its contents as expected)
def test_navbar(self):
    response1 = self.app.get("/")
    response2 = self.app.get("/selectPrediction")
    response3 = self.app.get("/selectSentiment")
    response4 = self.app.get("/fng")
    response5 = self.app.get("/portfolioSelector")

    # storing each response in a list and iterating over the list to apply th
    responses = [response1, response2, response3, response4, response5]

    # iterating over each response and apply the same tests to each (rather t
    for response in responses:
        # testing if navbar contents are there (should pass for all pages)
        assert b"CoinCast" in response.data
        assert b"Home" in response.data
        assert b"Coin Forecast" in response.data
        assert b"Market Sentiment" in response.data
        assert b"Crypto Tools" in response.data
        assert b"Fear & Greed Index" in response.data
        assert b"Portfolio" in response.data
        assert b"Search for crypto news..." in response.data

        # testing response status
        self.assertEqual(response.status, "200 OK")
        self.assertNotEqual(response.status, "100 Multiple Choice")
        self.assertNotEqual(response.status, "300 Multiple Choice")
        self.assertNotEqual(response.status, "400 Bad Request")
        self.assertNotEqual(response.status, "404 Not Found")
        self.assertNotEqual(response.status, "500 Internal Server Error")
```

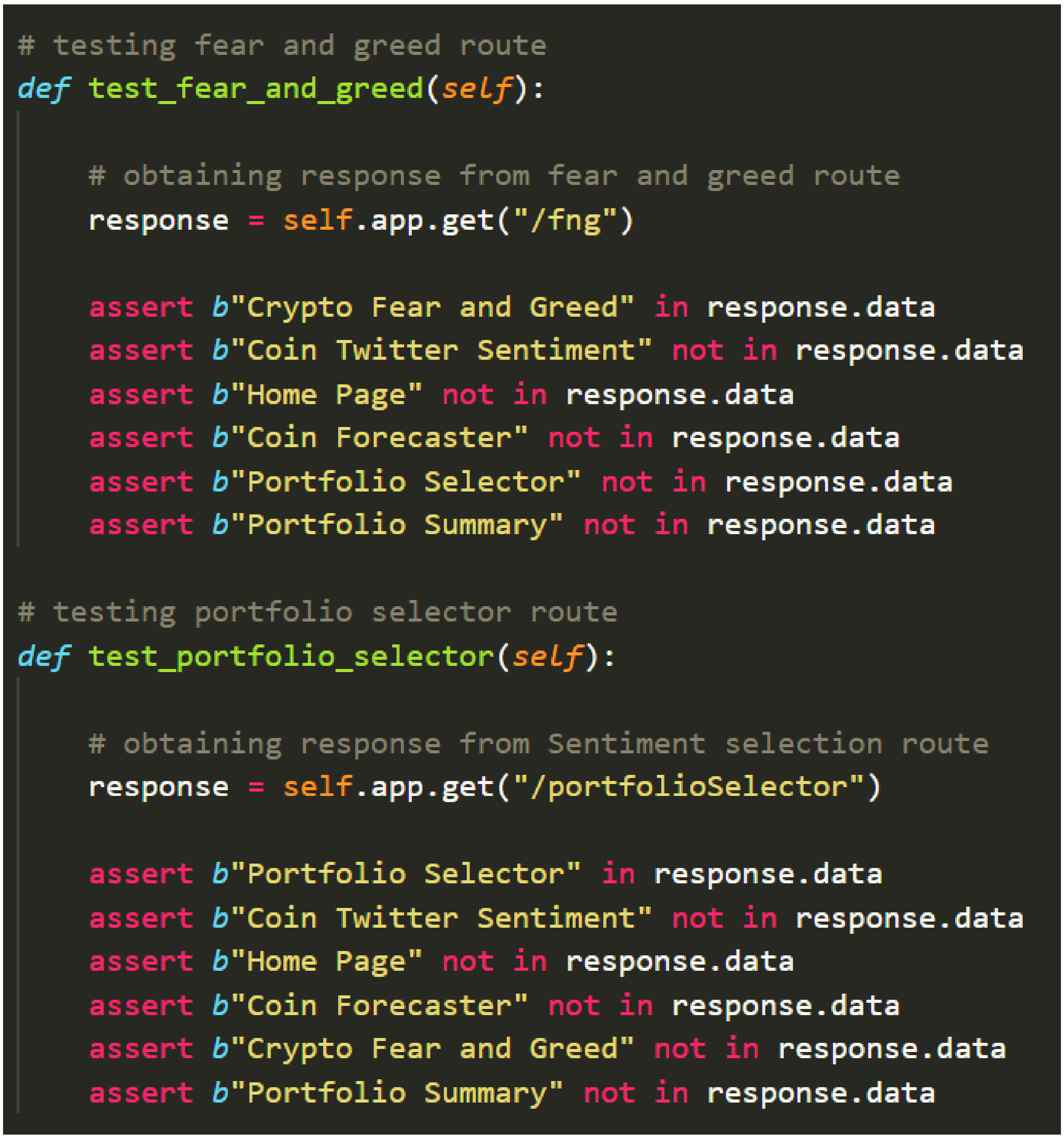
The following code snippets show how the contents of the home page, prediction selector, sentiment selector, portfolio selector, and the fear and greed pages are tested. It simply tests if the header in the HTML corresponds to the route itself so that we know we are on the correct page.



```python
# testing home route
def test_home(self):

    # obtaining response from home page route
    response = self.app.get("/")

    # ensuring the header corresponds to the page we're on an
    # (e.g if user in the home page, the page header should s
    assert b"Home Page" in response.data
    assert b"Coin Forecaster" not in response.data
    assert b"Coin Twitter Sentiment" not in response.data
    assert b"Crypto Fear and Greed" not in response.data
    assert b"Portfolio Selector" not in response.data
    assert b"Portfolio Summary" not in response.data

# testing prediction selection route
def test_prediction_selector(self):

    # obtaining response from prediction selection route
    response = self.app.get("/selectPrediction")

    assert b"Coin Forecaster" in response.data
    assert b"Home Page" not in response.data
    assert b"Coin Twitter Sentiment" not in response.data
    assert b"Crypto Fear and Greed" not in response.data
    assert b"Portfolio Selector" not in response.data
    assert b"Portfolio Summary" not in response.data
```

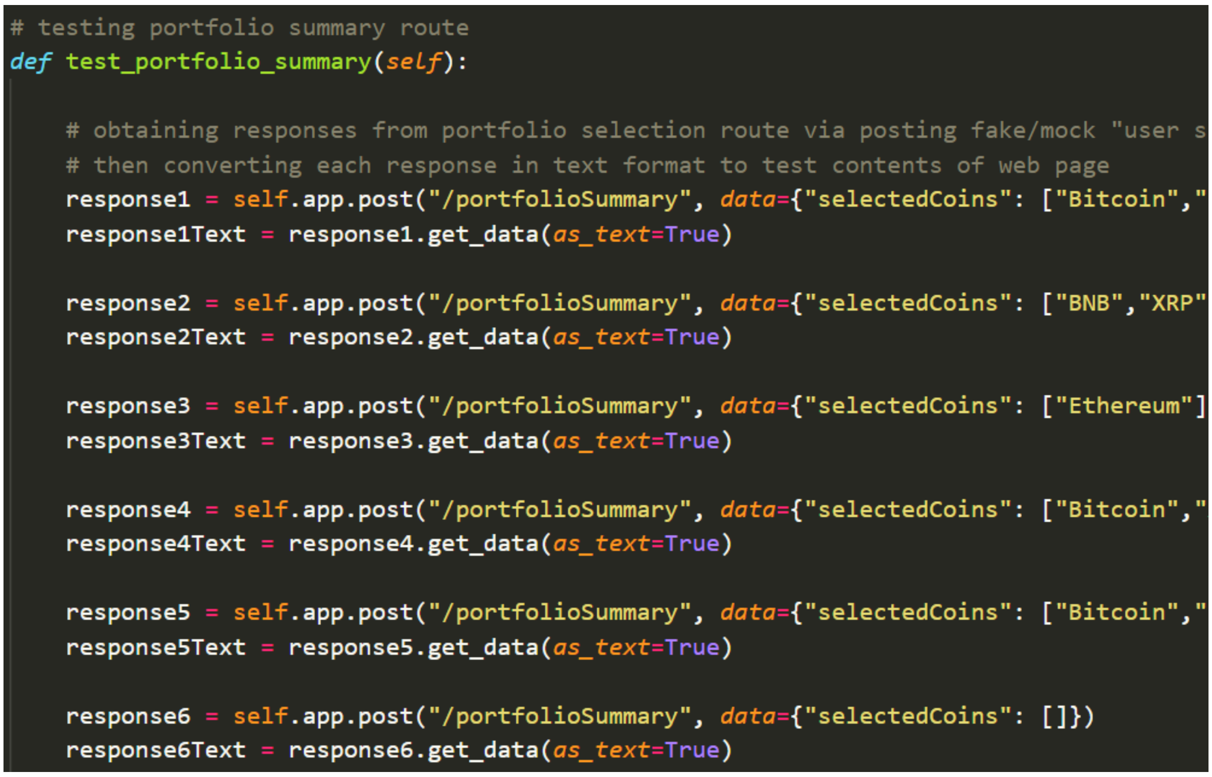


```python
# testing Sentiment selection route
def test_sentiment_selector(self):

    # obtaining response from Sentiment selection route
    response = self.app.get("/selectSentiment")

    assert b"Coin Twitter Sentiment" in response.data
    assert b"Home Page" not in response.data
    assert b"Coin Forecaster" not in response.data
    assert b"Crypto Fear and Greed" not in response.data
    assert b"Portfolio Selector" not in response.data
    assert b"Portfolio Summary" not in response.data
```

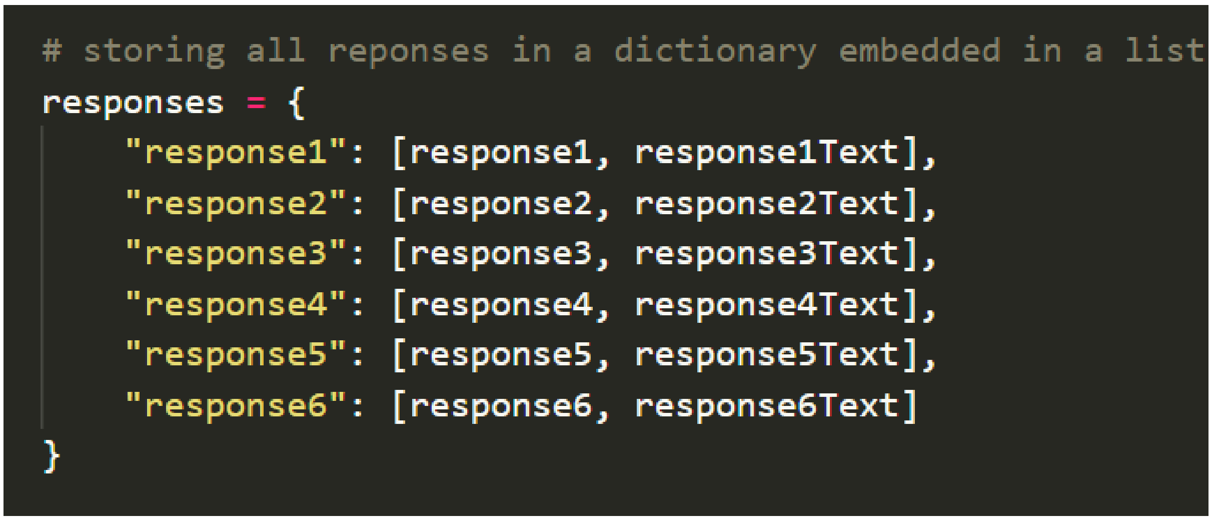


```python
# testing fear and greed route
def test_fear_and_greed(self):

    # obtaining response from fear and greed route
    response = self.app.get("/fng")

    assert b"Crypto Fear and Greed" in response.data
    assert b"Coin Twitter Sentiment" not in response.data
    assert b"Home Page" not in response.data
    assert b"Coin Forecaster" not in response.data
    assert b"Portfolio Selector" not in response.data
    assert b"Portfolio Summary" not in response.data

# testing portfolio selector route
def test_portfolio_selector(self):

    # obtaining response from Sentiment selection route
    response = self.app.get("/portfolioSelector")

    assert b"Portfolio Selector" in response.data
    assert b"Coin Twitter Sentiment" not in response.data
    assert b"Home Page" not in response.data
    assert b"Coin Forecaster" not in response.data
    assert b"Crypto Fear and Greed" not in response.data
    assert b"Portfolio Summary" not in response.data
```

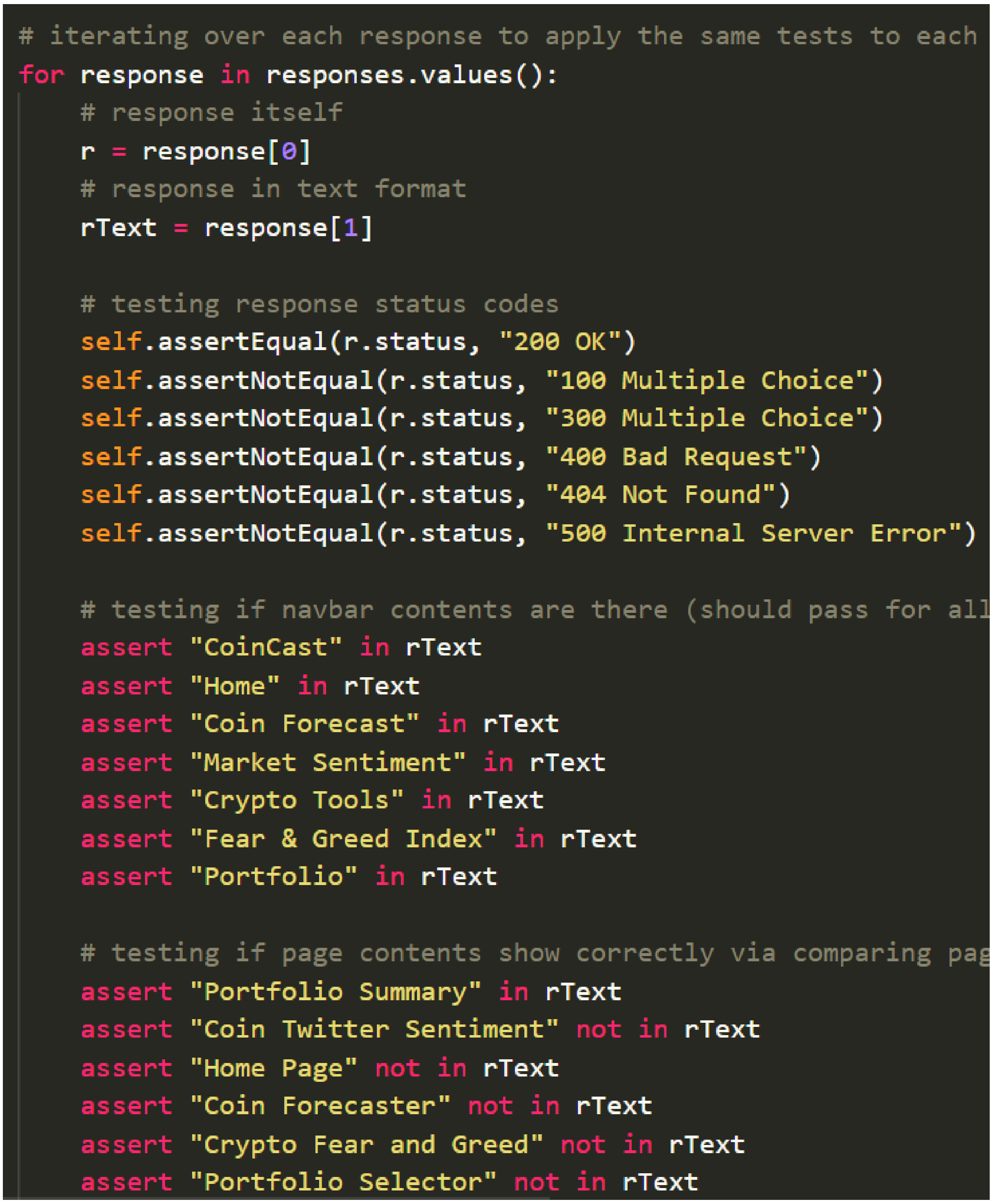
To access the portfolio summary page, it is required to have inputs saved for the portfolio page to display the user-selected coins. Because we used Flask sessions, we passed in hardcoded user-selected cryptocurrencies in order for the application to think that a portfolio already exists for each of the responses. Similar to before, we pass the responses to a data structure after which we iterate through each one and apply the same tests to it, ensuring the contents of the navbar are present and that the correct header is present.



```python
# testing portfolio summary route
def test_portfolio_summary(self):

    # obtaining responses from portfolio selection route via posting fake/mock "user s
    # then converting each response in text format to test contents of web page
    response1 = self.app.post("/portfolioSummary", data={"selectedCoins": ["Bitcoin","
    response1Text = response1.get_data(as_text=True)

    response2 = self.app.post("/portfolioSummary", data={"selectedCoins": ["BNB","XRP"
    response2Text = response2.get_data(as_text=True)

    response3 = self.app.post("/portfolioSummary", data={"selectedCoins": ["Ethereum"]
    response3Text = response3.get_data(as_text=True)

    response4 = self.app.post("/portfolioSummary", data={"selectedCoins": ["Bitcoin","
    response4Text = response4.get_data(as_text=True)

    response5 = self.app.post("/portfolioSummary", data={"selectedCoins": ["Bitcoin","
    response5Text = response5.get_data(as_text=True)

    response6 = self.app.post("/portfolioSummary", data={"selectedCoins": []})
    response6Text = response6.get_data(as_text=True)
```

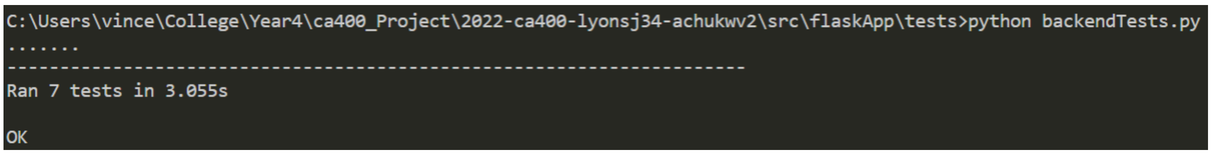


```python
# storing all reponses in a dictionary embedded in a list
responses = {
    "response1": [response1, response1Text],
    "response2": [response2, response2Text],
    "response3": [response3, response3Text],
    "response4": [response4, response4Text],
    "response5": [response5, response5Text],
    "response6": [response6, response6Text]
}
```



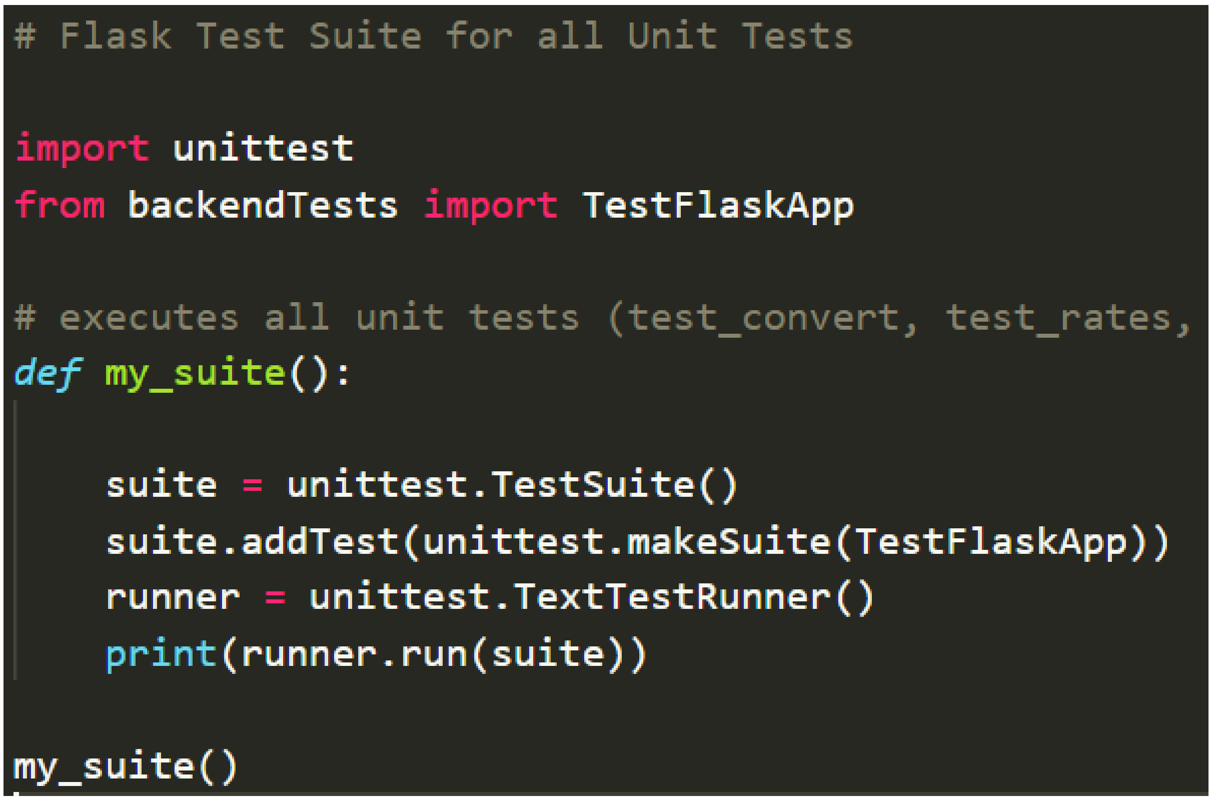
```python
# iterating over each response to apply the same tests to each
for response in responses.values():
    # response itself
    r = response[0]
    # response in text format
    rText = response[1]

    # testing response status codes
    self.assertEqual(r.status, "200 OK")
    self.assertNotEqual(r.status, "100 Multiple Choice")
    self.assertNotEqual(r.status, "300 Multiple Choice")
    self.assertNotEqual(r.status, "400 Bad Request")
    self.assertNotEqual(r.status, "404 Not Found")
    self.assertNotEqual(r.status, "500 Internal Server Error")

    # testing if navbar contents are there (should pass for all
    assert "CoinCast" in rText
    assert "Home" in rText
    assert "Coin Forecast" in rText
    assert "Market Sentiment" in rText
    assert "Crypto Tools" in rText
    assert "Fear & Greed Index" in rText
    assert "Portfolio" in rText

    # testing if page contents show correctly via comparing pag
    assert "Portfolio Summary" in rText
    assert "Coin Twitter Sentiment" not in rText
    assert "Home Page" not in rText
    assert "Coin Forecaster" not in rText
    assert "Crypto Fear and Greed" not in rText
    assert "Portfolio Selector" not in rText
```

Upon running the backend unit test, shown below was the output.



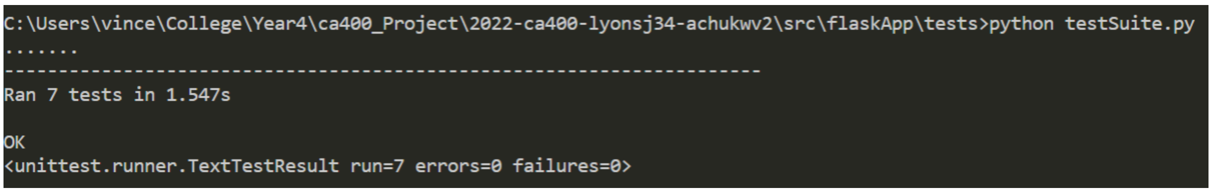
```
C:\Users\vince\College\Year4\ca400_Project\2022-ca400-lyonsj34-achukwv2\src\flaskApp\tests>python backendTests.py
.......
----------------------------------------------------------------------
Ran 7 tests in 3.055s

OK
```

The test suite aggregates unit tests that must be executed concurrently. The test runner arranges the execution of tests and presents the outcome to the user. Here, the test suite executes the unit tests from TestFlaskApp.



```python
# Flask Test Suite for all Unit Tests

import unittest
from backendTests import TestFlaskApp

# executes all unit tests (test_convert, test_rates,
def my_suite():

    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestFlaskApp))
    runner = unittest.TextTestRunner()
    print(runner.run(suite))

my_suite()
```

As shown below, the results of those tests are outputted to the console.

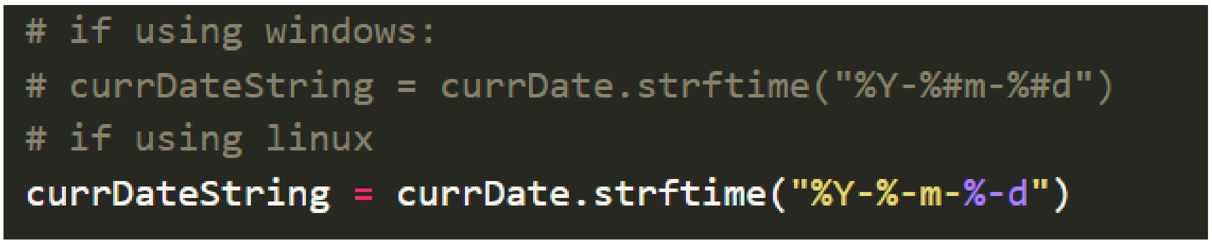


```
C:\Users\vince\College\Year4\ca400_Project\2022-ca400-lyonsj34-achukwv2\src\flaskApp\tests>python testSuite.py
.......
----------------------------------------------------------------------
Ran 7 tests in 1.547s

OK
<unittest.runner.TextTestResult run=7 errors=0 failures=0>
```

In backendTests.py, there was an issue when testing **test_sentiment_summary()** which adopted a similar approach and testing technique as **test_portfolio_summary()**. We left it in there just to show what we were trying to achieve when passing inputs to a route that can only be accessed when inputs are present. The tweepy API would reach the rate limiter when executing the unit test so we couldn't quite get that unit test working the way we wanted.

Ad-hoc Testing

We would both test the application during and after the implementation to identify any errors. We looked at certain sections of code and test different scenarios to ensure that all functionality worked as expected. For instance, depending on whether one of us would use Linux or Windows when developing the application, we would have to ensure the correct syntax was used for the date-time format when accessing the fear and greed index.



```
# if using windows:
# currDateString = currDate.strftime("%Y-%#m-%#d")
# if using linux
currDateString = currDate.strftime("%Y-%-m-%-d")
```

As explained in the code, if developing/deploying on a Linux or a Windows-like environment, it would be required to uncomment one line of code and comment out the other since the syntax for removing leading zeros in a date in Python varies (https://www.kite.com/python/answers/how-to-remove-leading-0's-in-a-date-in-python). The purpose of removing the leading zeros is because the fear and greed index API requires a date without the leading zeros.
We also tested opening the web application on different browsers such as chrome and microsoft edge. This didnt give us any errors. We also tested the resizing of windows to make sure that the application fit into any size window. At first we had issues with the html pages loading at set pixel sizes which we fixed by using % instead. This allows for the resizing of any of the pages in the web application

User testing of the arima model

For this we tested the arima model prediction over the space of 10 days at the same time each day. First we took the actual price for the first day and recorded it. We then ran the arima model and set it to one day. We then waited until the next day to see how close our prediction was. We continued to do this for the duration of our experiment phase and recorded the following:

Forecast error:

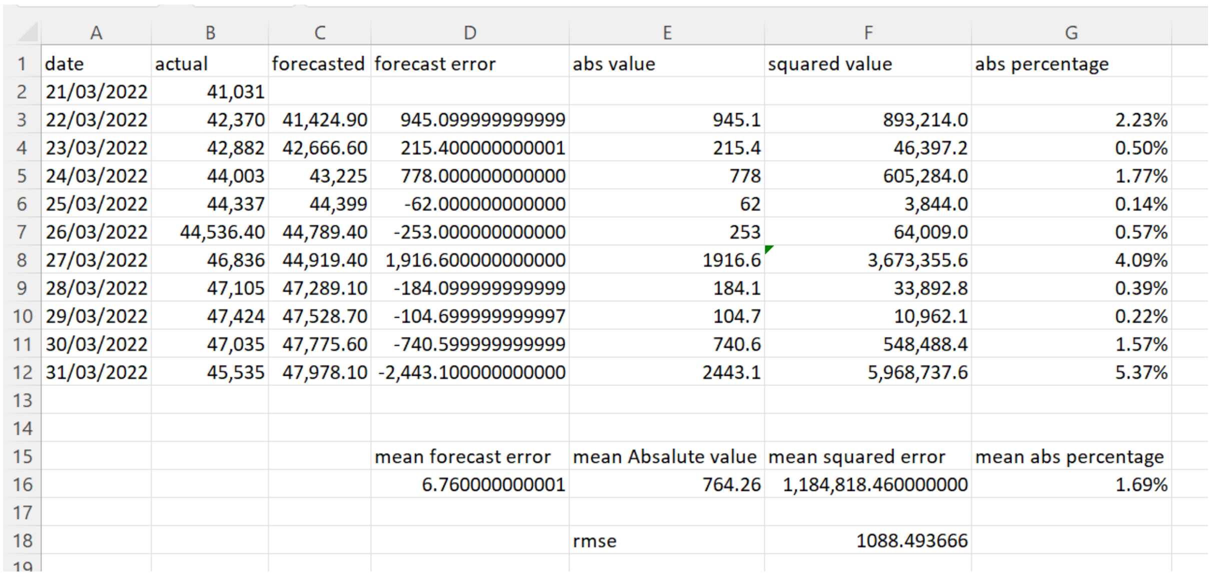
The difference between the actual price and the forecasted price. We then got the mean forecast error.

Absolute error:

This allowed us to use multiple functions that wouldnt work with minus values.

Percentage difference:

The difference between the forecasted and actual data. Seemed to work well when price movement was small, didnt pick up on trend flipping straight away. We noticed it would take a day for it to see that the direction in which the price is moving changed. This lend to weak prediction outputs when there was a change in direction. As you can see on the 26th and on the 31st with margins of 4% and 5%.



	A	B	C	D	E	F	G
1	date	actual	forecasted	forecast error	abs value	squared value	abs percentage
2	21/03/2022	41,031					
3	22/03/2022	42,370	41,424.90	945.099999999999	945.1	893,214.0	2.23%
4	23/03/2022	42,882	42,666.60	215.400000000001	215.4	46,397.2	0.50%
5	24/03/2022	44,003	43,225	778.000000000000	778	605,284.0	1.77%
6	25/03/2022	44,337	44,399	-62.000000000000	62	3,844.0	0.14%
7	26/03/2022	44,536.40	44,789.40	-253.000000000000	253	64,009.0	0.57%
8	27/03/2022	46,836	44,919.40	1,916.600000000000	1916.6	3,673,355.6	4.09%
9	28/03/2022	47,105	47,289.10	-184.099999999999	184.1	33,892.8	0.39%
10	29/03/2022	47,424	47,528.70	-104.699999999997	104.7	10,962.1	0.22%
11	30/03/2022	47,035	47,775.60	-740.599999999999	740.6	548,488.4	1.57%
12	31/03/2022	45,535	47,978.10	-2,443.100000000000	2443.1	5,968,737.6	5.37%
13							
14							
15				mean forecast error	mean Absalute value	mean squared error	mean abs percentage
16				6.760000000001	764.26	1,184,818.460000000	1.69%
17							
18					rmse	1088.493666	
19							

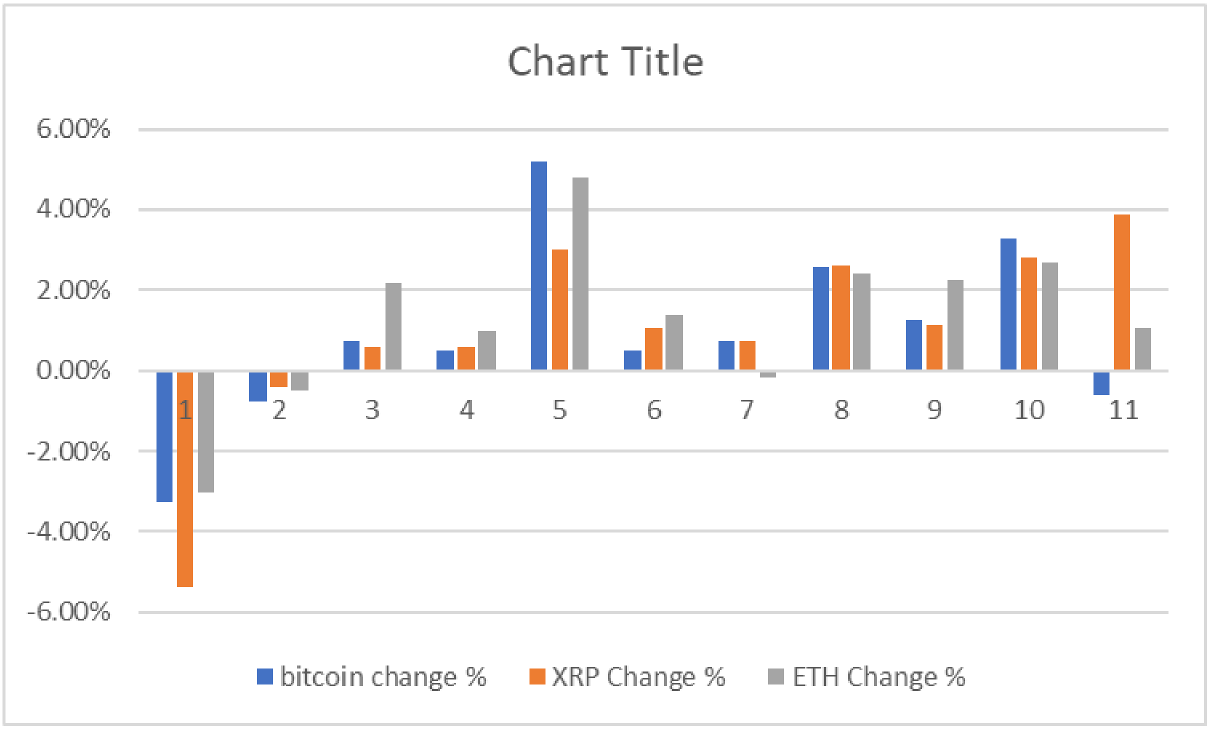
Results Based on our testing data:

We noticed that the market was very volatile. It was hard to predict trends in the data based on previous days as the cryptocurrency market can react so quickly to real life events. This meant that our arima model would always be a step behind in terms of prediction. We also noticed trends that smaller coins seem to always follow the trend of bitcoin. This meant that our enfacise if we were to do this project again would be getting our bitcoin prediction more accurate because most coins followed the same market trends as bitcoin. Below are some trends that we documented to solidify our reasoning behind this.

bitcoin change %	**XRP Change %**	**ETH Change %**
-3.29%	-5.38%	-3.05%
-0.79%	-0.42%	-0.48%
0.73%	0.59%	2.18%
0.52%	0.59%	0.98%
5.19%	2.99%	4.80%
0.49%	1.07%	1.37%
0.72%	0.75%	-0.18%
2.57%	2.61%	2.41%
1.27%	1.13%	2.24%
3.29%	2.80%	2.70%

-0.62% 3.88% 1.05%

That is the actual differences between the opening and closing prices of each coin over an 11 day period.



Here you can see the opening and closing differences for bitcoin xrp and etherum. As you can see they all seem to follow the same trend except for on day 11 were xrp went up alot more than the other. This could be due to market sentiment being differenent or news being leaked.

Possible Improvements :

If we were to do this project again we would definitely focus more on getting a more accurate prediction model. Running more testing early on would of identified trends in which our model lacked accuracy. We could have implemented a CI/CD pipeline in which we could of ran more tests through. More unit tests could of worked well in terms of what i displaying on the screen to the user. This would of eliminated alot of manual testing that we done and freed up some time. In future we would have ran our tests over the prices trends over a longer period of time to see if there are any other trends that we could have found. This may have helped with our prediction accuracy.