Project 03 Instructions

DSCI 11000: Introduction to Data Science

Description of Project

In this project, you will be asked to create and apply a class called `Stock`. This class is intended to provide tools for simulating the price of a stock with given characteristics over time.

Part A: Create the Stock Class.

In this part of the project, you will define a class called `Stock`. This class will contain four methods:

- `__init__()` – This is the constructor. It will initialize a handful of class attributes when an instance of the class is created.
- `simulate()` – This will generate a sequence of simulated daily stock prices for the stock, over a given period of time. We will call such a sequence of simulated prices a **run.**
- `annual_yield()` – This method calculates the annual yield (or rate of return) for the most recent simulated run for the stock.
- `monte_carlo()` – This method will generate several simulated runs for the stock, and will store the final price for each of the runs. This implements a process called **Monte Carlo simulation**. The purpose of Monte Carlo simulation is to generate many possible outcomes of a random process to get a sense as to what sort of behavior you might expect from the process.

The four methods you will need to create are described in more detail below.

`__init__()`

The constructor should take four parameters: `self`, `price`, `rate`, and `vol`.

- `price` is the current price of the stock.
- `rate` is the annual yield rate that the stock is expected to generate. In other words, it governs how quickly the price of the stock is expected to increase.
- `vol` is the volatility of the yield rate. It measures the uncertainty in the yield rate. Stocks with a low volatility will grow at approximately the same rate every day. Stocks with a high volatility will see their yield rates vary considerably from day to day.

The constructor should initialize values for the following seven parameters.

- `self.price` – This stores the current price of the stock. It should be set to `price`.
- `self.rate` – This stores the expected yield rate for the stock. It should be set to `rate`.
- `self.vol` – This store the volatility of the stock. It should be set to `vol`.
- `self.run` – Daily prices for the most recent simulated run. It should initially be set to `[price]`.
- `self.final` – The final price in the most recent simulated run. Initially, it should be set to `price`.
- `self.mc_prices` – This contains a list of final prices for each of several runs generated by the `monte_carlo()` function. It should initially be set to be an empty list.
- `self.mc_yields` – This contains a list of annual yield rates for each of several runs generated by the `monte_carlo()` function. It should initially be set to be an empty list.

`simulate()`

This method is intended to simulate a single run of the stock. It should take two parameters: `self`, and `days`. The method should simulate the changes in the stock process of a number of days provided by the `days` parameter. The process for doing so is as follows:

1. Replace `self.run` with a list containing a single element, which is equal to the current price of the stock. This is necessary to replace the results of any previous run that might have been created.
2. Write a loop that executes a number of times equal to `days`, and does the following each time it executes:
    a. It randomly generates a daily return according to a normal distribution. You can generate this with the following line of code.
    `ret = np.random.normal(self.rate/365, self.vol/365, 1)[0]`
    The attributes `self.rate` and `self.vol` are divided by 365 in the code above to convert from annual rates to daily rates. (In truth, there are only about 253 trading days a year on the Stock Exchange, but we will ignore this fact for the sake of simplicity.)

    b. Calculate a new daily price according to the following mathematical formula:
    $$NewPrice = MostRecentPrice \cdot e^{ret}$$
    You can use the function **`math.exp()`** to calculate e^{ret}.

    c. Append the new daily price to `self.run`.

3. When the loop finishes, store the final daily price in `self.final`.
4. Convert `self.run` to a numpy array with: **`self.run = np.array(self.run)`**.
5. Return `self.run`.

`annual_yield()`

This method calculates and returns the annual yield rate for the stock under the most recent simulated run. It takes only one parameter: `self`.

The mathematical formula for calculating the annual yield rate is: $yield = \ln\left(\frac{Final\ Price}{Original\ Price}\right) \cdot \frac{365}{days}$.

In the equation above, the variable **days** is equal to the number of days over which the run was simulated. Note that this will be **one less** than the length of the run. (Can you see why?)

You can use the function **`math.log()`** to calculate the natural logarithm, **ln()**. Note that many programming languages, Python included, use the expression **`log()`** to refer to the natural logarithm $\log_e()$, rather than the log base 10, $\log_{10}()$, as it typical in mathematics.

`monte_carlo()`

This method will generate several simulated runs of the stock, recording the final result of each run. It should take three parameters: `self`, `days`, and `n`. The parameter days will indicate the number of `days` that should be used to generate each run, and the parameter `n` will indicate the number of runs to generate. This method should perform the following steps.

1. Reset the attributes `self.mc_prices` and `self.mc_yields` to be empty lists.
2. Write a loop that executes `n` times. Each time the loop executes, it should perform the following steps:
    a. Simulate a run of length `days`. Note that you don't need to store the return value anywhere. All relevant information will be store in the class instance, `self`.
    b. Append the final price for the run into `self.mc_prices`.
    c. Append the annual yield rate for the run into `self.mc_yields`. You should use the `annual_yield()` method for this.
3. When the loop finishes, convert `self.mc_prices` and `self.mc_yields` into numpy arrays using code similar to what was provided in Step 4 of the `simulate()` method.
4. Return `self.mc_prices`.

Part B: Test the Class

Create an instance of the class `Stock` called `test_stock`. The stock should be created so that the current price is **100**, the expected annual yield is **0.06**, and the volatility is **0.5**.

Write a loop that executes 5 times, each time performing the following steps:

1. Simulate a run of the stock over a period of 200 days. You don't need to store the return value anywhere all of the information we need is stored in the attributes of `test_stock`.
2. Plot the simulated stock price over time by passing the simulated run to the function `plt.plot()`.
3. Store the annual yield for that run in a list called `annual_yields`.

When you are finished, call `plt.show()` to display the plot. Then print the list `annual_yields`.

You should see plots for five simulated runs for the price of the stock over the next 200 days. Run this cell a few times to get a sense of the variety of possible behaviors of this stock.

Part C: Rate and Volatility

In this part, you will explore how the **rate** and **vol** parameters affect the behavior of the stock. You should perform the following steps in this part of the project:

1. To ensure we all get the same results, set the seed to 1 using **np.random.seed(1)**.

2. Create two stocks: **stockA** and **stockB**. Both stocks should have a current price of 100 and an expected yield rate of 0.12. The volatility of **stockA** should be set to 0.3, and the volatility of **stockB** should be set to 0.8.

3. Call the **monte_carlo()** method for each of the two stocks, generate 1000 runs of each stock, each with a length of 150 days.

4. Print two sentences that say the following:

   ```
   Average Annual Yield for A over 1000 runs: 0.1212
   Average Annual Yield for B over 1000 runs: 0.1205
   ```

   Round the output to 4 decimal places, as shown above.

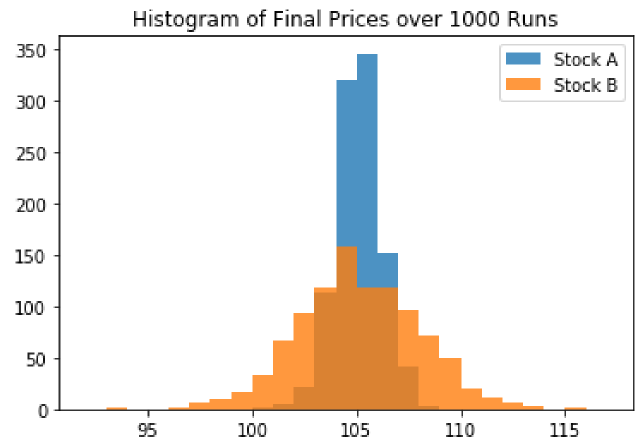
   To calculate the average yield rate, you can use the function **np.mean()**. For instance, to calculate the mean yield rate of several runs resulting from a Monte Carlo simulation, you could use the following code: **np.mean(stockA.mc_yields)**

5. Execute the following code to display histograms of the final prices for each of the runs generated by the Monte Carlo simulation for both Stock A and Stock B:

   ```
       plt.hist(___, bins=range(92,118), alpha=0.8, label='Stock A')
       plt.hist(___, bins=range(92,118), alpha=0.8, label='Stock B')
       plt.title('Histogram of Final Prices over 1000 Runs')
       plt.legend()
       plt.show()
   ```

   Replace the blanks in the code above with the prices generated by the Monte Carlo simulations.

   You should end up with a plot that looks something like this:



   Note that the simulated final prices tend to be around 105 (on average) for both stocks. However, the potential range of final prices is much more spread out for Stock B than for Stock A. This is because Stock B has a higher volatility.

Part D: Generating Specific Outcomes

The primary purpose of this part of the project is to illustrate how you can use seeds to “fix” a random process into a specific outcome.

Start by creating three stocks: `stockA`, `stockB`, and `stockC`. These stocks would have the following properties:

- Stock A has a current price of 78, an expected annual yield of 0.04, and a volatility of 1.2.
- Stock B has a current price of 75, an expected annual yield of 0.08, and a volatility of 0.8.
- Stock C has a current price of 72, an expected annual yield of 0.16, and a volatility of 0.6.

Under these conditions, Stock A there is a roughly 50% chance that Stock A will have the highest price after 200 days. Stocks B and C each have about a 25% chance of having the highest price after 200 days. We wish to find simulated examples of each of these situations occurring.

Run the following code with a positive integer (any positive integer) replacing the blank.

```
np.random.seed(__)
plt.plot(stockA.simulate(200), label="Stock A")
plt.plot(stockB.simulate(200), label="Stock B")
plt.plot(stockC.simulate(200), label="Stock C")
plt.legend()
plt.show()
```

Change the value of the seed until you obtain an example in which Stock A has the highest final simulated price of the three stocks. Then find a value of the seed that would result in Stock B having the highest final simulated price, and a value of the seed that would result in Stock C having the highest final simulated price.

Ultimately, you should have three copies of the code above, each one with a different seed and a different stock having the largest final price.

Hint: The desired seed values can be found through trial and error. But to help you find seed values that satisfy the condition you want, it might be helpful to use the following code to randomly generate and print seeds:

```
sd = np.random.choice(range(0,10000))
print(sd)
np.random.seed(sd)
plt.plot(stockA.simulate(200), label="Stock A")
plt.plot(stockB.simulate(200), label="Stock B")
plt.plot(stockC.simulate(200), label="Stock C")
plt.legend()
plt.show()
```

Once you have found a seed value that accomplishes what you want, delete the first two lines and manually set the seed value to the one that generated the desired results.

To ensure that everyone does this part on their own, **I want each person to have their own unique seed values on this part**. When you have found three seed values that generate the desired results, **please email them to me**. I will keep a list of the seeds have been “claimed” and will make them available to the other students. I will deduct points if you use seed values claimed by another student, or if you and another student have the same values, but neither of you claimed them before submitting your assignment.

Part E: Comparing two Stocks

To ensure we all get the same results on this part, set the seed to 1 using **`np.random.seed(1)`**.

Start by creating two stocks: **`stockA`** and **`stockB`**. These stocks would have the following properties:

- Stock A has a current price of 120, an expected annual yield of 0.07, and a volatility of 0.2.
- Stock B has a current price of 120, an expected annual yield of 0.05, and a volatility of 1.1.

Note that the two stocks have the same current price. Stock A has a higher expected yield than Stock B, and is much less volatile.

Run Monte Carlo simulations for both **`stockA`** and **`stockB`**. The simulations should generate 2000 runs over 365 days for each stock. For the 2000 run generated. calculate the following variables:

- prop_A_better – This should store the percentage of runs for which Stock A had a higher simulated final price than Stock B.
- prop_A_above – The percentage of runs for which Stock A had a simulated final price greater than 130.
- prop_B_above – The percentage of runs for which Stock B had a simulated final price greater than 130.
- prop_A_below – The percentage of runs for which Stock A had a simulated final price less than 120.
- prop_B_below – The percentage of runs for which Stock B had a simulated final price less than 120.

Hint: Since we stored the results of the Monte Carlo simulation as arrays, we can use Boolean subsetting to easily calculate the proportions above, **WITHOUT USING ANY LOOPS**. To see how this works, consider the following example:

```
arrayA = np.array([3, 6, 1, 4, 6])
arrayA = np.array([6, 2, 3, 1, 7])

b1 = arrayA > 5
b2 = arrayA > arrayB
```

When this code finishes, we will have:

- `b1 = [False, True, False, False, True]`
- `b2 = [False, True, False, True, False]`

If we wanted to count the number of `True` values in either `b1` or `b2`, we could use the `sum()` function.

When you are finished calculating the desired proportions, print the following output:

```
Proportions of Runs in which...
-------------------------------
A beats B: 0.6457

A ends above 130: 0.1744
B ends above 130: 0.2784

A ends below 120: 0.0
B ends below 120: 0.1864
```

Round your numbers to 4 decimal places, as shown above.

This output shows that Stock A outperforms Stock B about 65% of the time. However, since Stock B is more volatile than Stock B, it is more likely to result in a large final price (great than 130). The high volatility of the stock also results in Stock B having a higher likelihood than Stock A of losing value and ending up with a price less than 120.