Robo Advisor Project - Automated Testing Challenges

> Prerequisite: "Testing 1, 2, 3" Exercise

Setup

From within your project's virtual environment (e.g. "stocks-env"), install the `pytest` package:

```
conda activate stocks-env # for example, if necessary
pip install pytest # (first time only)
```

After writing tests, you should be able to run them from the root directory of your project repository:

```
pytest
```

Instructions

Read your existing project code. Think about ways to improve its readability and documentation. Think about ways to simplify and remove duplication. Think about decomposing the major responsibilities into component stand-alone functions (or maybe classes, if you prefer that kind of thing).

Think about the user experience and expectations. What do they need the program to do in order to consider it in "working condition"? What should the program do? Think of ways to express its desired functionality using common language.

Think of ways to verify your code is behaving as expected. Implement automated tests, for example using the `pytest` package. Feel free to reference the examples below, but it is not necessary to adhere to them exactly.

Challenges

> NOTE: the testing prompts below are examples to help you think about what kind of functionality to test. Ultimately everyone's programs may operate differently. The overall goal is just to implement tests in your program, in whatever way best makes most sense for your individual circumstance. 😸

Formatting Prices

Refactor price-formatting logic into a function called something like `to_usd()`, and implement a corresponding test called something like `test_to_usd()`.

Test various scenarios to ensure the price formatting function displays a dollar sign, two decimal places, and a thousands separator.

Compiling Request URLs

Refactor request URL compilation logic into a function called something like `compile_url()`, and implement a corresponding test called something like `test_compile_url()`.

Test to ensure the function accepts a stock symbol input parameter, and constructs the expected request url.

> SECURITY NOTE: because the URL will contain an API key parameter and we don't want to hard-code that value into our expected URL, it's OK to:
>
> A) abbreviate this test so it checks whether or not the compiled URL contains certain expected sub-strings, like the base URL and the given stock symbol, respectively, or
>
> B) interpolate / concatenate the `ALPHAVANTAGE_API_KEY` environment variable value into the expected URL

Issuing API Requests

Refactor API request logic into a function called something like `get_response()`, and implement a corresponding test called something like `test_get_response()`.

Test to ensure the function returns the expected response data in a usable format (i.e. a dictionary with keys `"Meta Data"` and `"Time Series (Daily)"`).

Processing API Responses

Refactor API data-processing logic into one or more functions called something like `parse_response()`, `calculate_recent_high()`, etc. Then implement corresponding test(s) called something like `test_parse_response()`, `test_calculate_recent_high()`, etc.

Test to ensure these functions operate as expected to further process the raw data into more usable structures and/or the final outputs themselves.

Writing to CSV

Refactor CSV file-writing logic into a function called something like `write_to_csv()`, and implement a corresponding test called something like `test_write_to_csv()`.

Test to ensure the function processes the provided data (the entire parsed response structure, or just the list of daily dictionaries), creates a new CSV file, and writes the data there. Optionally test to ensure the CSV file contains the proper headers and/or expected values.