

# Assignment 5: Stock Trading System using Python

Programming Languages SWE3006\_41

2022310853 Younwoo Park

Dept. of Systems Management Engineering  
Sungkyunkwan University

May 21, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overall Design . . . . .	3
1.2	Directory Structure . . . . .	3
<b>2</b>	<b>Stock Trading System Menus</b>	<b>4</b>
2.1	Registration . . . . .	4
2.2	Login . . . . .	4
2.3	Logout . . . . .	5
<b>3</b>	<b>Main Screen Menus</b>	<b>6</b>
3.1	View . . . . .	6
3.2	Buy/Sell Stocks . . . . .	6
3.2.1	Action Enum . . . . .	6
3.2.2	Validation and Storage . . . . .	6
3.2.3	Buy Stocks . . . . .	7
3.2.4	Sell Stocks . . . . .	8
3.3	View Portfolio . . . . .	9
3.3.1	Structure . . . . .	9
3.3.2	Implementation . . . . .	9
3.4	View Transaction History . . . . .	10
3.4.1	Structure . . . . .	10
3.4.2	Storage . . . . .	10
3.4.3	Implementation . . . . .	10
3.5	Auto Trading . . . . .	11
3.5.1	Implementation . . . . .	11
3.6	Logout . . . . .	12
<b>4</b>	<b>Market Module</b>	<b>12</b>
4.1	Stock Class . . . . .	12
4.1.1	Attributes . . . . .	12
4.1.2	Methods . . . . .	12
4.2	Market Class . . . . .	13
4.2.1	Attributes . . . . .	13
4.2.2	Methods . . . . .	13
4.3	Threading and Locking . . . . .	14
4.4	Auto-Trading Strategy Integration . . . . .	14
4.5	Conclusion . . . . .	14
<b>5</b>	<b>Strategy Module</b>	<b>14</b>
5.1	Strategy Class . . . . .	14
5.2	Random Strategy . . . . .	14
5.3	Momentum Strategy . . . . .	14
5.4	Moving Average Strategy . . . . .	14
<b>6</b>	<b>JSON Storage Structure</b>	<b>15</b>
6.1	Market Data . . . . .	15
6.2	User Data . . . . .	15
6.3	Transaction History . . . . .	15
<b>7</b>	<b>Miscellaneous</b>	<b>16</b>
7.1	Graceful Keyboard Interrupt Handling . . . . .	16
7.2	Python Version . . . . .	16

# 1 Introduction

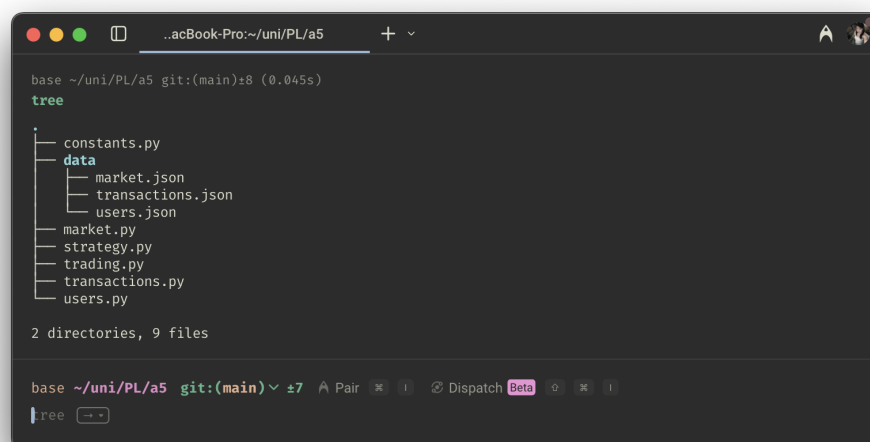
The goal of this assignment is to implement a stock trading system using Python. The system should allow users to buy and sell stocks, view their portfolio, and check the current stock prices. Users use a simple command-line interface to interact with the system. The system should also support multiple users, each with their own portfolio and trading history with a simple JSON-based storage architecture.

## 1.1 Overall Design

My implementation separates the system into several modules, each responsible for a specific part of the functionality. The main components are:

- **Market:** Manages stock prices and updates them periodically.
- **Users:** Manages user accounts, portfolios, and trading history.
- **Transactions:** Handles the logging of buy/sell transactions.
- **Strategy:** Implements different trading strategies for auto-trading.
- **Trading:** The main entry point for the program, handling user interactions and orchestrating the other components.
- **Constants:** Defines configuration constants, file paths, and enums for strategy and action types.
- **JSON Storage:** Uses JSON files to persist market data, user accounts, and transaction history.

## 1.2 Directory Structure

A terminal window titled '..acBook-Pro:~/uni/PL/a5' with a '+' icon and a window control bar. The terminal shows the command 'tree' and its output. The output shows a directory structure with a 'data' subdirectory containing three JSON files, and several Python files at the root level. The status bar at the bottom shows 'base ~/uni/PL/a5 git:(main) v ±7' and other icons.

```
base ~/uni/PL/a5 git:(main)±8 (0.045s)
tree
.
├── constants.py
├── data
│   ├── market.json
│   ├── transactions.json
│   └── users.json
├── market.py
├── strategy.py
├── trading.py
├── transactions.py
└── users.py

2 directories, 9 files

base ~/uni/PL/a5 git:(main) v ±7 A Pair x i @ Dispatch Beta @ x i
tree
```

Figure 1: Directory Structure

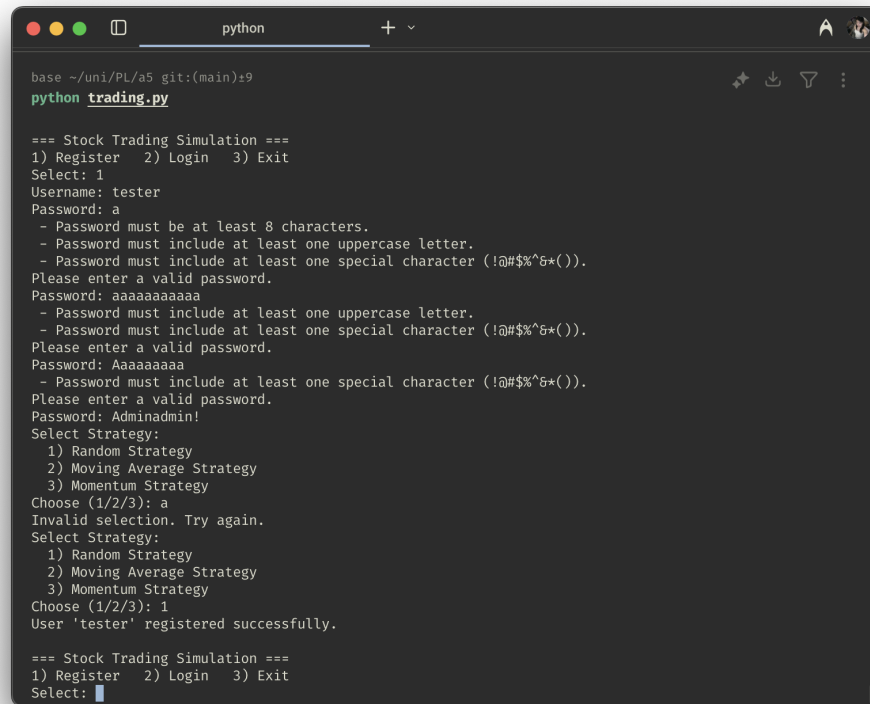
The `data/` directory contains JSON files for storing market data, user accounts, and transaction history. The main Python modules are responsible for managing the logic and functionality of the trading system.

Note: `trading.py` is equivalent to `2022310853_Trading.py`; just named in `trading.py` for simplicity while implementing and testing.

## 2 Stock Trading System Menus

### 2.1 Registration

The registration process takes place in `user.py`, where `register` is defined. Upon registration, users provide a username, password, and select a trading strategy. The system checks for duplicate usernames—case-sensitive and space-sensitive—and stores user data in `users.json`. Error Messages are displayed for invalid inputs.



```
base ~/uni/PL/a5 git:(main)±9
python trading.py

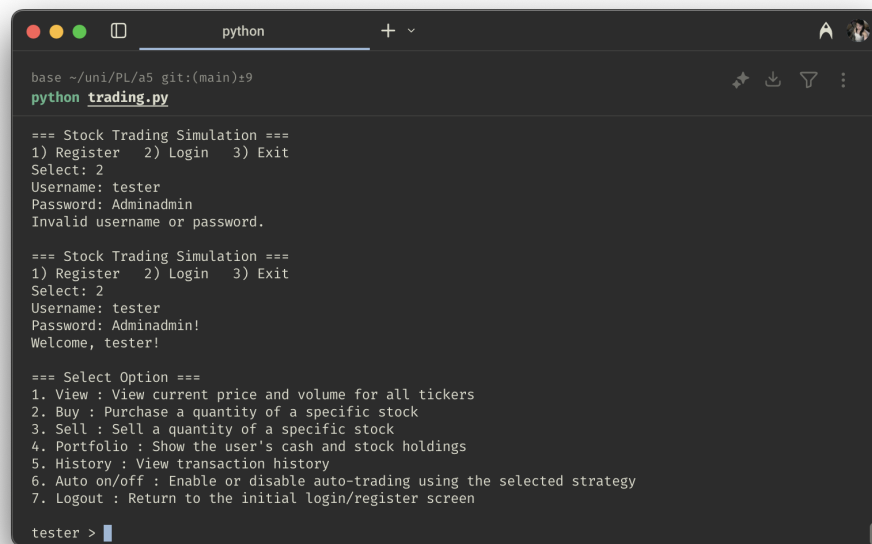
=== Stock Trading Simulation ===
1) Register 2) Login 3) Exit
Select: 1
Username: tester
Password: a
- Password must be at least 8 characters.
- Password must include at least one uppercase letter.
- Password must include at least one special character (!@#$%^&*()).
Please enter a valid password.
Password: aaaaaaaaaa
- Password must include at least one uppercase letter.
- Password must include at least one special character (!@#$%^&*()).
Please enter a valid password.
Password: Aaaaaaaaaa
- Password must include at least one special character (!@#$%^&*()).
Please enter a valid password.
Password: Adminadmin!
Select Strategy:
1) Random Strategy
2) Moving Average Strategy
3) Momentum Strategy
Choose (1/2/3): a
Invalid selection. Try again.
Select Strategy:
1) Random Strategy
2) Moving Average Strategy
3) Momentum Strategy
Choose (1/2/3): 1
User 'tester' registered successfully.

=== Stock Trading Simulation ===
1) Register 2) Login 3) Exit
Select: █
```

Figure 2: Example of registration process

### 2.2 Login

During login, the system verifies the username and password, which are case-sensitive and space-sensitive, against stored credentials. If the credentials are valid, the user is granted access to their portfolio and trading options. If not, an error message is displayed.



```
base ~/uni/PL/a5 git:(main)±9
python trading.py

=== Stock Trading Simulation ===
1) Register  2) Login  3) Exit
Select: 2
Username: tester
Password: Adminadmin
Invalid username or password.

=== Stock Trading Simulation ===
1) Register  2) Login  3) Exit
Select: 2
Username: tester
Password: Adminadmin!
Welcome, tester!

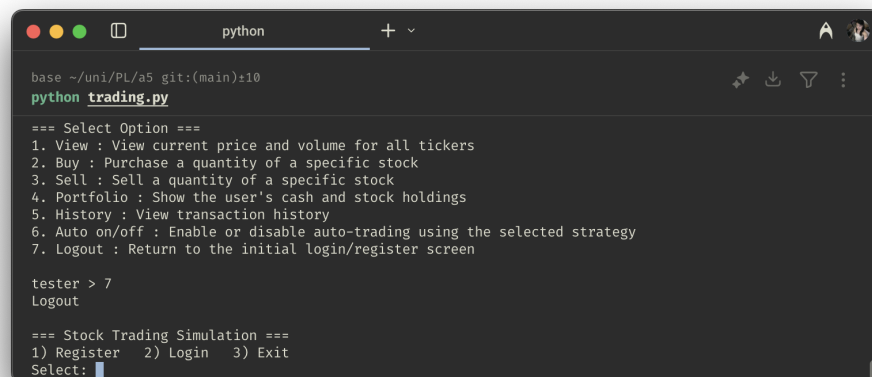
=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 
```

Figure 3: Example of login process

## 2.3 Logout

The logout process is straightforward. The user can choose to log out from the main menu, which will save their session data and close the program. The system ensures that all unsaved data is persisted before exiting through the `atexit` function in `trading.py`. This function calls `_cleanup()` to save the market data and user portfolios.



```
base ~/uni/PL/a5 git:(main)±10
python trading.py

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 7
Logout

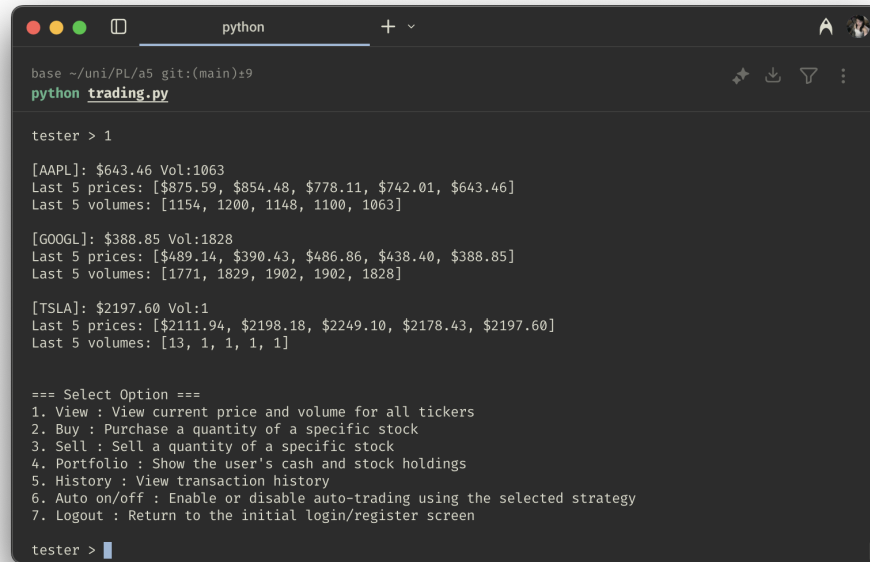
=== Stock Trading Simulation ===
1) Register  2) Login  3) Exit
Select: 
```

Figure 4: Example of logout process

## 3 Main Screen Menus

### 3.1 View

The first option on the main screen is to view the current stock prices. The system fetches the latest prices from `market.json` and displays them in a user-friendly format. Users can see the stock symbol, name, and current price.



```
base ~/uni/PL/a5 git:(main)±9
python trading.py

tester > 1

[AAPL]: $643.46 Vol:1063
Last 5 prices: [$875.59, $854.48, $778.11, $742.01, $643.46]
Last 5 volumes: [1154, 1200, 1148, 1100, 1063]

[GOOGL]: $388.85 Vol:1828
Last 5 prices: [$489.14, $390.43, $486.86, $438.40, $388.85]
Last 5 volumes: [1771, 1829, 1902, 1902, 1828]

[TSLA]: $2197.60 Vol:1
Last 5 prices: [$2111.94, $2198.18, $2249.10, $2178.43, $2197.60]
Last 5 volumes: [13, 1, 1, 1, 1]

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > |
```

Figure 5: Example of stock prices

### 3.2 Buy/Sell Stocks

The buy/sell stocks option allows users to trade stocks. Both buying and selling stock's underlying logic uses the same function, `execute_trade()` and `get_trade_options()`, which is defined in `user.py`. The system differentiates between buying and selling based on the user's input and uses a parameter to determine the action.

#### 3.2.1 Action Enum

The action is defined in `constants.py` as an enum, which makes the code more readable and maintainable.

```
class Action(Enum):
    BUY = "buy"
    SELL = "sell"
```

#### 3.2.2 Validation and Storage

The system checks if the user has sufficient funds or shares before proceeding with the transaction. The system also validates the stock symbol and price entered by the user. If the stock symbol is valid, the system updates the user's portfolio and transaction history accordingly.

The transaction history is stored in `transactions.json`, which keeps track of all buy/sell transactions made by users.

### 3.2.3 Buy Stocks

The buy stocks option allows users to purchase stocks. The system prompts the user for the stock symbol, quantity, and price. It checks if the user has sufficient funds and if the stock symbol is valid. If everything checks out, the transaction is processed, and the user's portfolio is updated.

```

python
base ~/uni/PL/a5 git:(main):10
python trading.py

tester > 2
===== Buy Menu =====
Available Cash: $6807.38
Your Holdings:
  AAPL: 2 shares @ avg 952.43
Enter ticker (['AAPL', 'GOOGL', 'TSLA']) or 'back' to return: back

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 2
===== Buy Menu =====
Available Cash: $6807.38
Your Holdings:
  AAPL: 2 shares @ avg 952.43
Enter ticker (['AAPL', 'GOOGL', 'TSLA']) or 'back' to return: none
Invalid ticker. Please try again.
Enter ticker (['AAPL', 'GOOGL', 'TSLA']) or 'back' to return: googl
Enter quantity to buy: -10
Invalid quantity. Please enter a positive integer.
Enter quantity to buy: a
Invalid quantity. Please enter a positive integer.
Enter quantity to buy: 10
Bought 10 GOOGL @ $45.39

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 2
===== Buy Menu =====
Available Cash: $6353.48
Your Holdings:
  AAPL: 2 shares @ avg 952.43
  GOOGL: 10 shares @ avg 45.39
Enter ticker (['AAPL', 'GOOGL', 'TSLA']) or 'back' to return: aapl
Enter quantity to buy: 100
Insufficient funds to buy 100 shares of AAPL.

```

Figure 6: Example of buying stocks

#### 1. User Input & Validation

The `buy_stock()` method begins by prompting the user for a stock ticker and quantity via `get_trade_options()`. It validates that: 1. The ticker exists (`DEFAULT_STOCK`), 2. The quantity is a positive integer.

#### 2. Trade Execution

1. The `execute_trade()` method is called with the action `ActionType.BUY`.
2. The system retrieves the latest price from the `Market` object.
3. It checks if the user has enough balance to cover the total cost ( $\text{price} \times \text{qty}$ ).

If valid the balance is reduced and the stock is added or updated in the portfolio.

New average price is calculated using a **weighted average** formula:

$$\text{new\_avg\_price} = (\text{old\_qty} * \text{old\_avg} + \text{new\_qty} * \text{price}) / \text{total\_qty}$$

### 3. Logging

The transaction is logged with `log_transaction()`. A confirmation message is printed unless the trade was triggered by auto-trading.

#### 3.2.4 Sell Stocks

The sell stocks option allows users to sell stocks from their portfolio. The system prompts the user for the stock symbol, quantity, and price. It checks if the user has enough shares to sell and if the stock symbol is valid. If everything checks out, the transaction is processed, and the user's portfolio is updated.



```
base ~/uni/PL/a5 git:(main) python trading.py

tester > 3
===== Sell Menu =====
Available Cash: $6353.48
Your Holdings:
  AAPL: 2 shares @ avg 952.43
  GOOGL: 10 shares @ avg 45.39
Enter ticker (dict_keys(['AAPL', 'GOOGL'])) or 'back' to return: back

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 3
===== Sell Menu =====
Available Cash: $6353.48
Your Holdings:
  AAPL: 2 shares @ avg 952.43
  GOOGL: 10 shares @ avg 45.39
Enter ticker (dict_keys(['AAPL', 'GOOGL'])) or 'back' to return: kasjd
Invalid ticker. Please try again.
Enter ticker (dict_keys(['AAPL', 'GOOGL'])) or 'back' to return: googl
Enter quantity to sell: -10
Invalid quantity. Please enter a positive integer.
Enter quantity to sell: a
Invalid quantity. Please enter a positive integer.
Enter quantity to sell: 14
You only have 10 shares of GOOGL.

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 3
===== Sell Menu =====
Available Cash: $6353.48
Your Holdings:
  AAPL: 2 shares @ avg 952.43
  GOOGL: 10 shares @ avg 45.39
Enter ticker (dict_keys(['AAPL', 'GOOGL'])) or 'back' to return: googl
Enter quantity to sell: 7
Sold 7 GOOGL @ $350.35
```

Figure 7: Example of selling stocks

#### 1. User Input & Validation

The `sell_stock()` method similarly asks the user for a ticker and quantity. It ensures two things: 1. The user **owns** the stock, 2. The quantity to sell is **less than or equal** to what's owned.

#### 2. Trade Execution

1. The `execute_trade()` method is called with `ActionType.SELL`.
2. It calculates proceeds as  $qty \times current\_price$ .
3. The user's balance is increased.



4. If all shares are sold, the stock is removed from the portfolio. Otherwise, the quantity is reduced.

### 3. Logging

Like buying, this action is recorded in `transactions.json`.

### 3.3 View Portfolio

Each User in the system maintains a personal **portfolio**, which is a dictionary mapping stock tickers (`str`) to **Holding** objects. This structure tracks both the quantity and average purchase price of each stock the user owns.

```

base ~/uni/PL/a5 git:(main)±10
python trading.py

tester > 4

Cash: $8805.93
AAPL: 2 @avg$952.43 -> $1089.38 (+14.38%)
GOOGL: 3 @avg$45.39 -> $584.24 (+1187.16%)
Total: $12737.41

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester >

```

Figure 8: Example of portfolio

#### 3.3.1 Structure

The user's portfolio is a dictionary:

```
portfolio: Dict[str, Holding] = field(default_factory=dict)
```

```

@dataclass
class Holding:
    qty: int
    avg_price: float

```

Each **Holding** stores:

- `qty`: total number of shares owned.
- `avg_price`: weighted average purchase price.

Since JSON does not support custom objects directly, this object is serialized/deserialized using `.to_dict()` and `.from_dict()` methods. This structure supports stock holdings and accurate computation profit/loss.

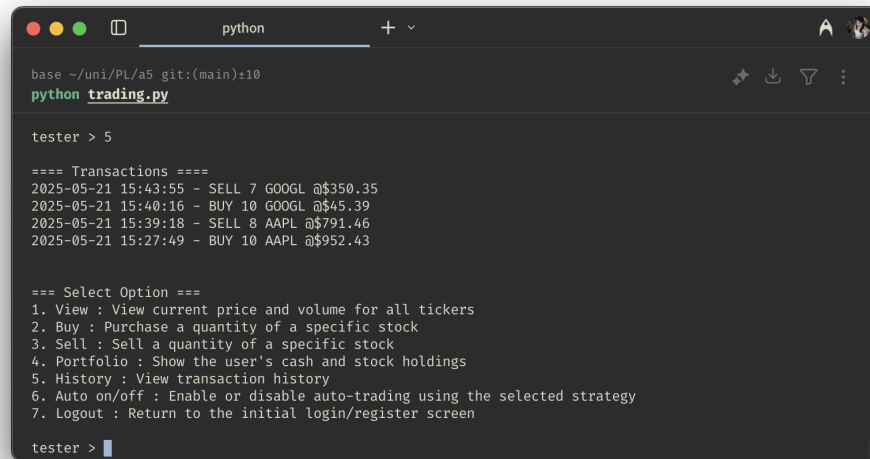
#### 3.3.2 Implementation

The `view_portfolio()` method in `user.py` retrieves the portfolio data and formats it for display. It shows the stock symbol, quantity owned, average purchase price, and current market value.

The portfolio is updated in real-time as users buy/sell stocks, updating the total quantity and average price accordingly.

### 3.4 View Transaction History

The transaction history option allows users to view their past transactions. The system retrieves the transaction history from `transactions.json` and displays it in a user-friendly format. Users can see the action (buy/sell), stock symbol, price, quantity, and timestamp of each transaction.



```
base ~/uni/PL/a5 git:(main)+10
python trading.py

tester > 5

==== Transactions ====
2025-05-21 15:43:55 - SELL 7 GOOGL @$350.35
2025-05-21 15:40:16 - BUY 10 GOOGL @$45.39
2025-05-21 15:39:18 - SELL 8 AAPL @$791.46
2025-05-21 15:27:49 - BUY 10 AAPL @$952.43

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 
```

Figure 9: Example of transaction history

#### 3.4.1 Structure

Each individual transaction is represented as an instance of the `Transaction` dataclass defined in `transaction.py`:

```
@dataclass
class Transaction:
    time: str
    ticker: str
    action: ActionType
    qty: int
    price: float
```

Each transaction can be serialized to a dictionary using the `to_dict()` to facilitate JSON storage.

#### 3.4.2 Storage

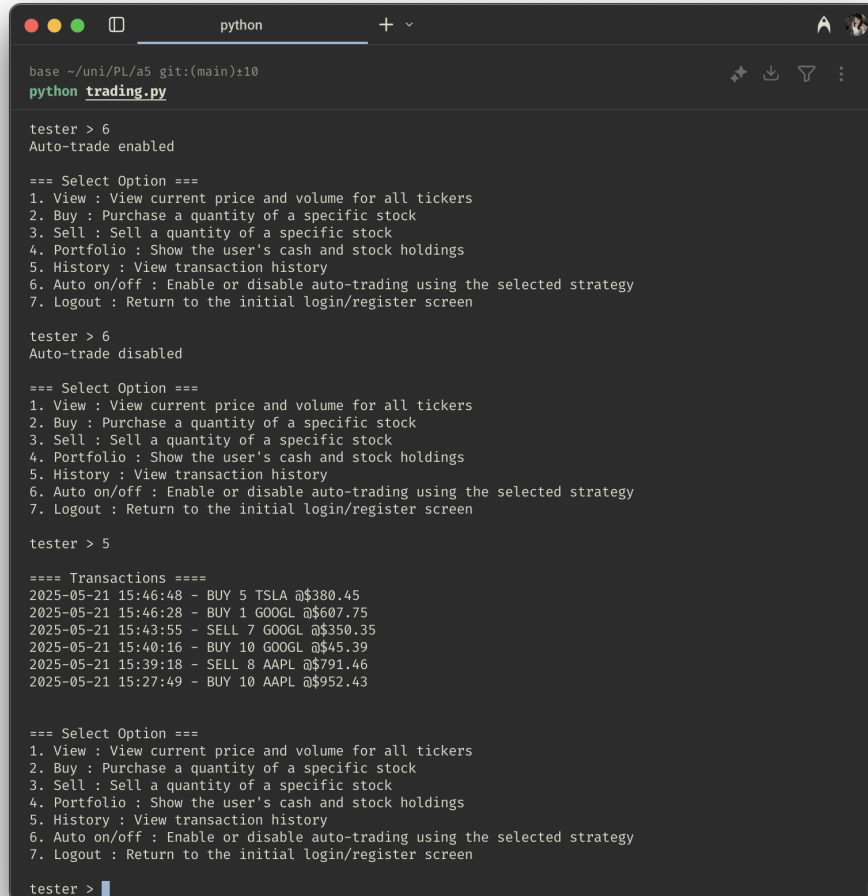
Transactions are saved in a file called `transactions.json`. The format is a dictionary where each key is a username, and each value is a list of transaction records for that user.

#### 3.4.3 Implementation

At program startup, the system loads existing transactions from `transactions.json` into memory. When a user performs a buy/sell action, the transaction is logged in this file. The `display_history()` method under `transaction.py` takes in the username and retrieves the transaction history for that user. It formats the data for display, showing the action (buy/sell), stock symbol, price, quantity, and timestamp of each transaction.

### 3.5 Auto Trading

Auto-trading allows users to let a predefined strategy automatically make buy or sell decisions on their behalf. Once enabled, the system makes trading decisions for them in the background based on market data updates.



```

base ~/uni/PL/a5 git:(main)+10
python trading.py

tester > 6
Auto-trade enabled

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 6
Auto-trade disabled

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 5

===== Transactions =====
2025-05-21 15:46:48 - BUY 5 TSLA @$380.45
2025-05-21 15:46:28 - BUY 1 GOOGL @$607.75
2025-05-21 15:43:55 - SELL 7 GOOGL @$350.35
2025-05-21 15:40:16 - BUY 10 GOOGL @$45.39
2025-05-21 15:39:18 - SELL 8 AAPL @$791.46
2025-05-21 15:27:49 - BUY 10 AAPL @$952.43

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester >

```

Figure 10: Example of auto trading

As shown in the image, new transactions are automatically logged in the transaction history. The user can also disable auto-trading at any time, which will stop the background trading activity.

This feature runs even when the user is logged out, as long as the market is open. The system will continue to execute trades based on the user's selected strategy until they log back in and disable it.

#### 3.5.1 Implementation

The auto-trading feature is implemented in the `Market` class (`market.py`) within the `_run_market()` method. This method continuously updates the market and checks for users with `auto = True`. If auto-trading is enabled, the system fetches the user's selected strategy from `strategy.py` and executes it using the latest market data.

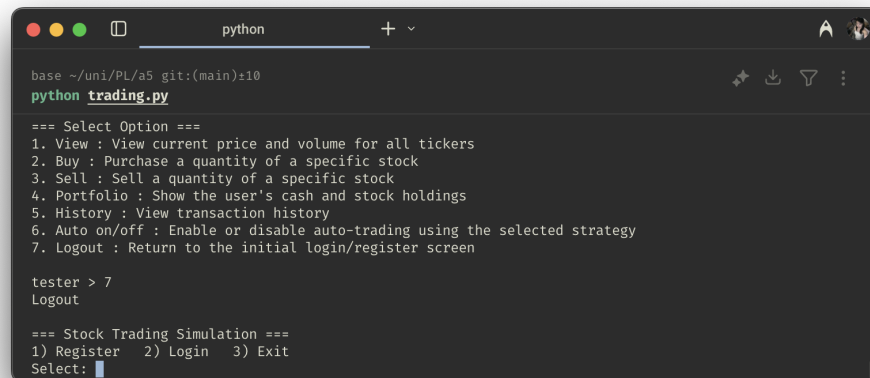
Supported strategies include `RandomStrategy`, `MomentumStrategy`, and `MovingAverageStrategy`, all of which inherit from a shared `Strategy` interface. These strategies make trading decisions by calling `user.execute_trade()` based on logic defined in their respective `execute()` methods.

The entire `_run_market()` method — including both market updates and strategy execution — runs in a single background thread started by the `open()` method using:

```
threading.Thread(target=self._run_market, daemon=True).start()
```

### 3.6 Logout

The logout option allows users to exit the program. The overall session data is saved when the program exits, not when the user logs out. After logging out, the user is returned to the main menu, where they can choose to log in again or exit the program.



```
base ~/uni/PL/a5 git:(main)±10
python trading.py

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > 7
Logout

=== Stock Trading Simulation ===
1) Register  2) Login  3) Exit
Select: █
```

Figure 11: Example of logout

## 4 Market Module

The market module—defined under `market.py`—is responsible for managing stock prices and updating them periodically. It defined two core components:

- **Stock**: a dataclass representing a single stock’s price and volume history.
- **Market**: a class that manages all stocks, periodically updates their values, and executes auto-trading logic for users with enabled strategies.

### 4.1 Stock Class

The `Stock` dataclass encapsulates the state and history of an individual stock.

#### 4.1.1 Attributes

- `ticker`: `str`: The stock symbol (e.g., “AAPL”).
- `history`: `list[dict]`: A list of dictionaries storing time-stamped price and volume data.

#### 4.1.2 Methods

- `to_dict()` -> `Dict`: Converts the stock’s history into a JSON-serializable format.
- `get_current_price()` -> `float` | `None`: Returns the most recent price in the history, or `None` if empty.
- `get_latest_volume()` -> `int` | `None`: Returns the most recent volume in the history.
- `update(price: float, volume: int, timestamp: str)` -> `None`: Appends a new record to the stock history and truncates it to the most recent `MAX_MARKET_HISTORY` entries.

This class is purely data-oriented and is used for both updating and serializing/deserializing market data.

## 4.2 Market Class

This class maintains a dictionary of all tradable stocks, runs the live market update loop, and integrates auto-trading functionality for users.

### 4.2.1 Attributes

- `stocks`: `Dict[str, Stock]`: Maps ticker symbols to `Stock` objects.
- `_running`: `bool`: Boolean flag indicating whether the market is currently active.
- `lock`: `threading.Lock`: Ensures thread-safe updates to market data.

### 4.2.2 Methods

`_load_market()` -> `None`

- Loads existing stock data from `market.json` if it exists.
- If not, initializes empty history for each ticker in `DEFAULT_STOCK` and writes it to file.
- Reconstructs the in-memory `stocks` dictionary from JSON.

`_save_market()` -> `None`

- Serializes the `stocks` dictionary to `market.json` by calling each `Stock`'s `to_dict()`.

`_update_market()` -> `None`

- Iterates over all stocks and updates their price and volume with random fluctuations.
- Appends the update to each stock's history.
- Saves updated state to disk.

`_run_market()` -> `None`

- Called inside a background thread once the market is opened.
- Executes `_update_market()` every `MARKET_UPDATE_INTERVAL` seconds.
- Also loops through all users (imported dynamically from the `users` module) and applies their respective auto-trading strategies if `user.auto` is enabled.
- Uses `STRATEGY_OPTIONS[user.strategy]` from `strategy.py` to retrieve the appropriate strategy instance.

`open()` -> `None`

- Initializes the market if not already running.
- Loads market data and starts the market thread.

`close()` -> `None`

- Stops the market and persists stock data.

`view()` -> `None`

- Prints the current price and volume of each stock.
- Also shows the last 5 prices and volumes to simulate a mini time series chart.

It fetches the latest stock prices from an external API and stores them in `market.json`. The module also provides functions to get the current price of a specific stock and to update the prices periodically.

The market module is implemented in `market.py`, which includes the following functions: - `get_stock_price(symbol)`: Fetches the current price of a stock from the API. - `update_stock_prices()`: Updates the prices of all stocks in the market. - `get_all_stock_prices()`: Returns a dictionary of all stock prices.

### 4.3 Threading and Locking

The `Market` uses `threading.Thread` to run live updates in the background, independent of user interaction. It also uses a `threading.Lock()` to guard `_update_market()` from race conditions when auto-trading executes user operations during updates.

### 4.4 Auto-Trading Strategy Integration

Auto-trading logic is built into `_run_market()`:

- After each update, it loops through all users who have `auto = True`.
- It retrieves and calls the `execute()` method of the user's strategy (`Random`, `Momentum`, or `MovingAverage`).
- The strategy then calls `user.execute_trade()` based on logic.

### 4.5 Conclusion

This module plays a central role in simulating stock trading. It provides a real-time, multi-threaded stock market engine with historical tracking and hooks for user strategies. Its architecture is cleanly modular, and the JSON-based data persistence ensures session continuity between runs.

## 5 Strategy Module

The strategy module is responsible for implementing different trading strategies that users can choose from. It defines three strategies: `Random`, `Momentum`, and `MovingAverage`. Each strategy is implemented as a class that inherits from a base class `Strategy`.

### 5.1 Strategy Class

The `Strategy` class is an abstract base class that defines the interface for all trading strategies. It contains only one method, `execute()`, which must be implemented by subclasses. This method takes a `Market` object and a `User` object as parameters and performs the trading logic.

### 5.2 Random Strategy

The `Random` strategy randomly decides whether to buy or sell a stock. It uses the `random` module to generate a random number and decides the action based on that. The `execute()` method takes a `Market` object and a `User` object as parameters and performs the trading logic.

### 5.3 Momentum Strategy

The `Momentum` strategy buys stocks that have been increasing in price over the last `MOMENTUM_PERIOD` days and sells stocks that have been decreasing in price. It uses the `get_latest_volume()` method of the `Market` class to get the latest volume of a stock and decides whether to buy or sell based on that. If not enough data is available, it skips the trade until more data is collected.

### 5.4 Moving Average Strategy

The `MovingAverage` strategy calculates the moving average of a stock's price over the last `MOVING_AVERAGE_PERIOD` days. It buys stocks that are currently above their moving average and sells stocks that are below their moving average. The `execute()` method takes a `Market` object and a `User` object as parameters and performs the trading logic. If not enough data is available, it skips the trade until more data is collected.

## 6 JSON Storage Structure

### 6.1 Market Data

The market data is stored in a JSON file called `market.json`. This file contains the current prices and volumes of all stocks in the market. The data is structured as follows:

```
{
  "AAPL": {
    "history": [
      {
        "time": "2024-05-20 20:00:00",
        "price": 135.55,
        "volume": 430
      },
      ...
    ]
  },
  "GOOGL": {
    "history": [... ]
  }
}
```

Each ticker contains a `history` list that maintains chronological market updates. This structure allows:

- Efficient appending of new market ticks.
- Easy access to current price and historical data.
- Direct serialization/deserialization via Python's built-in `json` module.

### 6.2 User Data

The user data is stored in a JSON file called `users.json`. This file contains the usernames, passwords, and portfolios of all users. The data is structured as follows:

```
{
  "user1": {
    "password": "hashed_password",
    "portfolio": {
      "AAPL": 10,
      "GOOGL": 5
    },
    "strategy": "Random",
    "auto": true
  },
  ...
}
```

### 6.3 Transaction History

The transaction history is stored in a JSON file called `transactions.json`. This file contains the details of all buy and sell transactions made by users. The data is structured as follows:

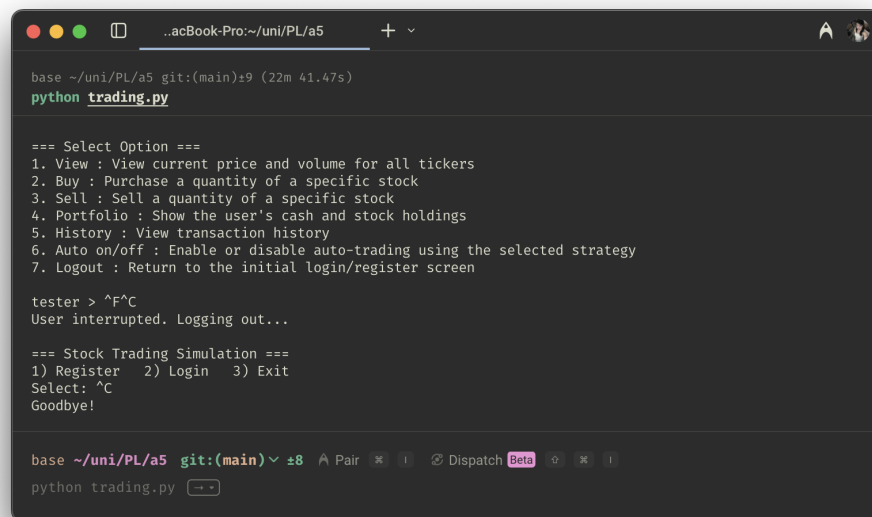
```
{
  "user1": [
    {
      "action": "buy",
```

```
"ticker": "AAPL",
"price": 135.55,
"volume": 10,
"timestamp": "2024-05-20 20:00:00"
},
...
],
...
}
```

## 7 Miscellaneous

### 7.1 Graceful Keyboard Interrupt Handling

The program handles keyboard interrupts (Ctrl+C) gracefully by catching the `KeyboardInterrupt` exception. This allows the user to exit the program without losing any unsaved data. The market is closed, and all data is saved before exiting.



```
base ~/uni/PL/a5 git:(main)±9 (22m 41.47s)
python trading.py

=== Select Option ===
1. View : View current price and volume for all tickers
2. Buy : Purchase a quantity of a specific stock
3. Sell : Sell a quantity of a specific stock
4. Portfolio : Show the user's cash and stock holdings
5. History : View transaction history
6. Auto on/off : Enable or disable auto-trading using the selected strategy
7. Logout : Return to the initial login/register screen

tester > ^F^C
User interrupted. Logging out...

=== Stock Trading Simulation ===
1) Register 2) Login 3) Exit
Select: ^C
Goodbye!

base ~/uni/PL/a5 git:(main)±8 A Pair * i @ Dispatch Beta * i
python trading.py
```

Figure 12: Example of graceful exit

### 7.2 Python Version

The code is implemented in Python 3.11.12. Because of the `None` type hinting, the code is not compatible with earlier versions of Python.