Trading Simulator

This simulation platform is built in Python OOP, which is designed to simulate a realistic trading environment using the SET50 daily ticks dataset. It supports core functionalities such as order matching, real-time data streaming, portfolio management, and fee calculations, providing a practical foundation for developing and testing algorithmic trading strategies.

Each trader is provided with an initial cash balance of 10,000,000 baht upon portfolio creation. This amount represents the trader's total available capital for the entire competition. Participants are expected to maximize profits by designing and implementing efficient trading algorithms to compete against other teams.

Feature contained in the simulation:

- Stock & portfolio management:
    - Initiate & Load portfolio information.
    - Create summarized portfolio information throughout the day of trading.
    - Update the stock market value along with a streaming system.
    - Calculation of the necessary information for creating strategies.
        - Amount Cost
        - Average cost of stock in the portfolio
        - Unrealized & Unrealized in percentage
        - Realized
        - NAV (Net Asset Value)
        - Max draws down
        - Calmar ratio
- Ordering & matching during the streaming of the trading market:
    - Creating an order for selling or buying stock in the market
        - Ordered stock must be in SET50 & volume required to be in full stock unit at least 100 volume (able to be mod by 100), and unable to order partial volume of stock
          For example, 100 volume is acceptable, but 150 (150 % 100 = 50) cannot be ordered.
        - Ordering stock must be in either Buy or Sell.
        - When an order buys stocks from the market, if the trader's portfolio doesn't have enough cash balance, the order will be rejected.
        - When an order sells stock from the market, if the portfolio of trade doesn't have enough volume of stock in its portfolio, the order will be rejected.
    - Matching the order with the market streaming situation:
        - Validated orders will be sent to the order book, which will check the streaming market and match the order in terms of volume and price with the market.

Installing dependency

Before you can run the trading simulator, you need to install the required dependencies. Follow the steps below to set up your Python environment properly:

1. Ensure Python 3.13 or higher is installed.
    a. (Optional) Install pyenv to handle multiple Python versions.
2. Download the `TradeSim` from here.
3. (Recommended) Create and activate a virtual environment for `TradeSim`.

```
cd path\to\TradeSim
python -m venv TradeSim-venv
.\TradeSim-venv\Scripts\activate
```

4. Install required dependencies using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

5. If the Jupyter kernel is running, restart the kernel.

Introduction to TradeSim

This notebook walks you through the usage of the TradeSim platform. You will learn how to load market data, register your team and strategy, run simulations, and view results. Follow each section to set up and test your trading algorithm correctly.

1. This code block loads the daily tick data used in the simulation. Please make sure to update the daily tick file provided daily **through Google Classroom**. Participants must ensure the file path points to the latest version before running the simulation.

```
#TODO: Change ticks information daily
daily_ticks = "./marketInfo/ticks/Daily_Ticks.csv"

df = pd.read_csv(daily_ticks)
df['TradeDateTime'] = pd.to_datetime(df['TradeDateTime'])
grouped = df.groupby('ShareCode')
```

2. The following block handles the **team name** and **strategy name** setup.
   Participants are required to assign their team and strategy names according to the specified format.
   - Names must contain only **alphanumeric characters**, **hyphens (-)**, or **underscores (_)**
   - Maximum length: **30**

```
#TODO: Replace with your team name
team_name = "example"
strategy_name = "example_strategy"

pattern = r'^[A-Za-z0-9-_]{1,30}$'
if not bool(re.match(pattern, team_name)) or not bool(re.match(pattern,
strategy_name)):
    raise ValueError("Team name or strategy name is invalid. Please use only
alphanumeric characters, hyphens, and underscores, with a maximum length of 30
characters.")

# Init trade system
trading_Sim = TradeSim.tradeSim(team_name)
strategy_runner = trading_Sim.get_strategy_runner()
```

3. This block handles the **import and initialization of the trading strategy**. If there is an error during import (e.g., the file or class is missing), it will be printed out, and the simulation will continue **without a strategy**. This allows participants to observe market behavior before implementing their strategy logic.
   For more information on implementing a strategy, please see Implement a Strategy.

```
try:
    strategy_module = importlib.import_module(f"strategy.{strategy_name}")
    importlib.reload(strategy_module)
except ImportError as e:
    active_strategy_fn = None
    print(f"Error in strategy module: {e}")

strategy_class = getattr(strategy_module, strategy_name, None)
```

4. This block runs the **full market simulation**, where each tick is streamed and processed in real-time. The simulation supports two execution modes:

   - `with_visual = True`:
     Runs in **visual mode**, simulating real-time streaming with delays, ideal for demonstration or debugging.
   - `with_visual = False`:
     Runs in **headless mode**, reduces the runtime of the simulation, ideal for faster testing and evaluation.

   Use the `with_visual` flag to toggle between modes based on your preference.

```
#TODO: Change the `with_visual` according to your preference
# True if you want to see the visual simulation
# False if you want to finish the simulation faster
with_visual = False
```

```
threads = []
latest_prices = {}
def stream_symbol(symbol, data):
...
trading_Sim.flushTransactionLog()
trading_Sim.create_transaction_summarize(team_name)
trading_Sim.save_portfolio()
trading_date = df['TradeDateTime'].dt.date.iloc[0]
trading_Sim.save_summary_csv(trading_date)
```

Implementing a Strategy

All custom strategies must be implemented inside the `strategy/` directory. You will find two important Python files:

- `Strategies_template.py`: the abstract base class you must inherit from
- `Example_strategy.py`: a basic example to help you get started

TODO: Create your strategy class; you need to inherit from `Strategies_template.py`

Strategies_template.py: The Base Class
This file defines the strategy interface. Your strategy must inherit from `Strategy_template` and implement the required `on_data()` method:

```
from abc import ABC, abstractmethod
from datetime import timedelta

class Strategy_template(ABC):
    def __init__(self,owner,strategy_name, handler):
        self.owner = owner
        self.strategy_name = strategy_name
        self.handler = handler

    @abstractmethod
    def on_data(self,row):
        """
        MUST be implement by subclass
        DO NOT EDIT THIS CLASS
        """
        pass
```

Example_strategy.py: A Sample Strategy
The following class is a simple example of a custom strategy that buys and sells selected stocks based on basic rules:

```
from strategy.Strategies_template import Strategy_template

class Example_strategy(Strategy_template):
    def __init__(self, handler):
        super().__init__("example", "Example_strategy", handler)
        self.buy_counts = {s: 0 for s in ["ADVANC", "AOT", "AWC", "BANPU", "BBL"]}
        self.sell_counts = {s: 0 for s in ["ADVANC", "AOT", "AWC", "BANPU", "BBL"]}
    def on_data(self, row):
        symbol = row['ShareCode']
        price = row['LastPrice']
        if not self.handler.check_port_has_stock(symbol, 100):
            self.handler.create_order_to_limit(100, price, "Buy", symbol)
        else:
            stocks = self.handler.get_stock_by_symbol(symbol)
            if stocks:
                buy_price = stocks[0].get_buy_price()
                if price < buy_price:
                    self.handler.create_order_to_limit(100, price, "Sell", symbol)
...
```

To ensure your strategy is loaded correctly, the following naming rules must be strictly followed

The `team_name` and `strategy_name` declared in your main script must match the arguments passed to `super().__init__()` inside your strategy class.

- The `team_name` and `strategy_name` are declared in `TradingSimulation.ipynb`.

```
#TODO: Replace with your team name
team_name = "example"
strategy_name = "example_strategy"
```

- `Team_name` must be the first parameter in the class constructor.
- `Strategy_name` must be the second parameter in the class constructor, and the class name.

```
class example_strategy(Strategy_template):
    def __init__(self, handler):
        super().__init__("example", "example_strategy", handler)
```

The `strategy_name` must also match the Python file name (without the .py extension)

A guide to create a buy/sell order

To place an order in your strategy, use the method provided by the `handler` object. There are two variations of the create order method you can use: `create_order_to_limit()` and `create_order_at_market()`

The parameter:

Parameter	Type	Description
volume	int	Must be a multiple of 100 (e.g., 100, 200).
price	float	The price at which you want to trade.
side	string	Either `"Buy"` or `"Sell"` (case-insensitive).
symbol	str	Stock symbol (e.g., `"AOT"`, must be in the SET50).

Create order to limit

Place a limit order at the specified price. A limit order will only be executed at the given price or better. Use this method when you want more control over the execution price.

```
create_order_to_limit(volume, price, side, symbol)
```

Example usage:

```
self.handler.create_order_to_limit(100, 35.67, "Buy", "AOT")
self.handler.create_order_to_limit(100, 39.89, "Sell", "PTT")
```

Create order at the market

Place a market order, which will be executed immediately at the best available price. Use this method when volume is more important than price precision.

```
create_order_at_market(volume, side, symbol)
```

Example usage:

```
self.handler.create_order_at_market(1000, "Buy", "AOT")
self.handler.create_order_at_market(5000, "Sell", "PTT")
```

Return value from Create order

Both `create_order_to_limit()` and `create_order_at_market()` return a string message that indicates whether the order was successfully placed or rejected. This allows your strategy to confirm the outcome of the order and handle errors appropriately.
Example of a valid order:

```
order_result = self.handler.create_order_to_limit(100, 58, "Buy", "AOT")
print(order_result)
```

Output:

```
Order for AOT (Vol: 100, Price: 58.0, Side: Buy) created successfully.
```

If the order is invalid, such as due to insufficient holdings or an incorrect volume, it will not be registered. Instead, an error message will be returned with the reason.
Example of invalid order:

```
order_result = self.handler.create_order_to_limit(500, 58, "Sell", "AOT")
print(order_result)
```

Output:

```
[ERROR] Order for AOT (Vol: 500, Price: 58.0, Side: Sell) skipped due to Cannot sell AOT as it
is not in the portfolio or insufficient volume.
```

Accessing information in the daily ticks row

The `row` parameter passed to your strategy's `on_data()` method represents a single tick (data point) for a stock symbol at a specific time**.**

Sample Structure of `row` :

```
{
    'ShareCode': 'EGCO',
    'TradeDateTime': Timestamp('2025-01-10 09:55:28'),
    'LastPrice': 113.0,
    'Volume': 7100,
    'Value': 802300.0,
    'Flag': 'BUY'
}
```

The data can be accessed as follows:

```
symbol = row["ShareCode"]
price = row["LastPrice"]
timestamp = row["TradeDateTime"]
volume = row["Volume"]
```

This allows you to build condition-based logic using real-time tick information.

The following are key data from the `row`:

Key	**Type**	**Description**
ShareCode	str	Stock symbol (e.g., “AOT”)
TradeDateTime	pandas.Timestamp	Exact datetime of the trade tick.
LastPrice	float	The most recent traded price of the stock at this moment.
Volume	int	Number of shares traded in this tick.
Value	float	Number of shares traded in this tick.
Flag	str	Market condition flag

Note: The `row` variable is set as **read-only** to prevent any value from being assigned.

Accessing Portfolio Data

The following methods are available for participants to inspect the current state of their portfolio. These can be used during or after the simulation to monitor holdings, cash balance, or export results.

Get information from the portfolio

```
get_portfolio_info()
```

A competitor can retrieve a summary of portfolio information from this method.

Return:

- Summary of information in portfolio

Example:

```
pprint(self.handler.get_portfolio_info())
```

Output:

```
{'Calmar Ratio': -17.836228391479434,
 'Cash Balance': 10074723.68,
 'Max Drawdown (%)': -28.44,
 'Max NAV': 10104164.72,
 'Min NAV': 7269444.2,
 'Net Asset Value': 10074723.68,
 'Number of Sells': 8,
 'Number of Stocks': 0,
 'Number of Wins': 8,
 'Owner': 'Testing',
 'Previous Day Max DD (%)': -28.44,
 'Realized P&L': 74723.68,
 'Relative Drawdown': -1.7145955018322022e-07,
 'Return rate': 507.31890610747763,
 'Total Cost': 0.0,
 'Unrealized %': 0.0,
 'Unrealized P&L': 0.0,
 'Win Rate': 100.0}
```

Get information from the stocks in the portfolio

```
get_all_stocks_info()
```

Competitors can retrieve information about their stocks that are in their portfolio from this method.

Return:

- Dicts of all stock information in the portfolio

Example:

```
pprint(self.handler.get_all_stocks_info())
```

Output:

```
[{'Symbol': 'AOT', 'Buy Price': 35.23, 'Actual Volume': 200, 'Average Cost': 0.0,
'Market Price': 35, 'Amount Cost': 0.0, 'Market Value': 0.0, 'Unrealized P&L': 0.0,
'Unrealized %': 0.0, 'Realized P&L': 0.0, 'Buy_time': '2025-06-23 11:27:53'}]
```

Get information from the stocks in the portfolio by symbol

```
get_stock_by_symbol(symbol)
```

Competitors can get specific stock by providing a preferred symbol of stock symbol.

Parameter:

- symbol: (*string*) - symbol

Return:

- List of stocks with the same symbol in the parameter

Example

```
print(self.handler.get_stock_by_symbol('AOT')[0].get_stock_info())
print(self.handler.get_stock_by_symbol('AOT')[1].get_stock_info())
```

Output:

```
{'Symbol': 'AOT', 'Actual Volume': 100, 'Buy Price': 30.2, 'Market Price': 30, 'Average
Cost': 0.0, 'Amount Cost': 0.0, 'Market Value': 0.0, 'Unrealized P&L': 0.0, 'Unrealized
%': 0.0, 'Realized P&L': 0.0, 'Buy_time': '2025-06-23 11:54:48'}
{'Symbol': 'AOT', 'Actual Volume': 100, 'Buy Price': 35.23, 'Market Price': 35,
'Average Cost': 0.0, 'Amount Cost': 0.0, 'Market Value': 0.0, 'Unrealized P&L': 0.0,
'Unrealized %': 0.0, 'Realized P&L': 0.0, 'Buy_time': '2025-06-23 11:54:48'}
```

Check if portfolio has stock in the provided symbol

```
has_stock(symbol, volume)
```

Competitor can check if their portfolio stock of preferred symbol and volume or not with this method

Parameter:

- symbol: (*string*) – symbol
- volume: (*integer*) – volume

Return

- Boolean (True or False)

Example

```
print(self.handler.has_stock('AOT',100))
```

Output

```
True
```

Additional getter method for strategy runner

The `self.handler` object provides several getter methods to help participants monitor their portfolio throughout the simulation. These methods can be used to retrieve information such as holdings, cash balance, NAV, drawdowns, and performance metrics.

Below is a list of available getter methods:

Portfolio Checks	
check_port_has_stock(symbol, volume)	Returns True if the portfolio holds at least a volume of the symbol
Portfolio Info	
get_owner()	Returns the team name
get_all_stocks_info()	Returns a list of summarized stock info, grouped by symbol.
get_portfolio_info()	Returns a summary of the current portfolio
get_stock_by_symbol(symbol)	Returns a specific stock object by symbol
get_total_stock_volume_by_symbol(symbol)	Returns the total number of shares held for a specific symbol
get_cash_balance()	Returns the current cash available in the portfolio
Performance Metrics	
get_roi()	Returns the portfolio's current Return on Investment
get_max_draw_down()	Returns the maximum drawdown encountered so far
get_number_of_wins()	Returns the total number of profitable trades executed
get_number_of_sells()	Returns the total number of sell transactions executed

Get information from stock in portfolio

The portfolio itself summarizes and totals all of the information from the stock. If a competitor would like to use more detailed stock information, competitors require to call it via a specific method.

```
get_stock_info()
```

competitor can call this method to get all of the information about the stock

Return:

- Dicts of stock information

Example:

```
print(self.handler.get_stock_by_symbol('AOT')[0].get_stock_info())
print(self.handler.get_stock_by_symbol('ADVANCE')[0].get_stock_info())
```

Output:

```
{'Symbol': 'AOT', 'Actual Volume': 100, 'Buy Price': 30.2, 'Market Price': 30, 'Average Cost': 0.0, 'Amount
Cost': 0.0, 'Market Value': 0.0, 'Unrealized P&L': 0.0, 'Unrealized %': 0.0, 'Realized P&L': 0.0,
'Buy_time': '2025-06-23 12:42:14'}
{'Symbol': 'ADVANC', 'Actual Volume': 100, 'Buy Price': 280.87, 'Market Price': 279, 'Average Cost': 0.0,
'Amount Cost': 0.0, 'Market Value': 0.0, 'Unrealized P&L': 0.0, 'Unrealized %': 0.0, 'Realized P&L': 0.0,
'Buy_time': '2025-06-23 12:42:14'}
```

Keep in mind that stock that has been bought can only be accessed via `get_stock_info_by_symbol()` method as shown in the example.

Once competitors can access stock in the portfolio, these following getter methods are now available to be used.

Additional getter method for stock

The following methods are available to retrieve detailed information about a stock position held in the portfolio. These can be useful for analysis, logging, or debugging strategy during simulation.

Basic Info	
`get_symbol()`	Returns the stock symbol (e.g., "AOT", "PPT")
Trade Details	
`get_buy_price()`	Returns the price per share at which the stock was purchased
`get_start_vol()`	Returns initial number of shares bought
`get_actual_vol()`	Returns remaining shares currently held
`get_buy_time_str()`	Returns timestamp of the purchase in readable format
Valuation	
`get_mkt_price()`	Returns current market price of the stock
`get_amount_cost()`	Returns total cost spent on this stock (buy_price × volume)
`get_market_value()`	Returns current market value of the held stock (market_price × volume)
Performance	
`get_unrealized()`	Returns current unrealized profit/loss
`get_unrealized_in_percentage()`	Returns unrealized profit/loss as a percentage
`get_realized()`	Returns realized profit/loss (from partial sells)

Penalties for Unauthorized Modification of the Trading System

To ensure fairness and integrity in this trading simulation, the following rules and penalties apply:

Prohibited Actions
Any developer found to have **intentionally modified any part of the trading system** other than the designated strategy section for the purpose of gaining an unfair advantage or bypassing system rules **will be subject to penalties**, including:

- Modifying order execution logic
- Altering portfolio/account handling mechanisms
- Changing the data feed or timing
- Interfering with result tracking or logging processes
  Any code manipulation that affects global behavior or evaluation outcomes

Intention and Scope
If our team detects intentional modification aimed at benefiting a specific team by cheating or bypassing the agent rules of the competition:

- **Your team's simulation results will be disqualified and not counted in the final evaluation.**
- **Your strategy will be removed from the pool of valid entries.**

Permitted Area
You are allowed to:

- Modify or implement your strategy **within the designated strategy file that was inherited from strategy_template.py**
- Use public APIs and allow utility/helper functions that do not compromise the system's fairness

Final Decision

All decisions regarding disqualification and result invalidation are **final and at the discretion of the organizing team**.