

Write an algorithmic trading app for stocks and stock options

with SQL relational database and a web based user interface supporting multiple internal users with different roles and permissions

Subsequent changes will be entered as

Begin mmddyyyy-#

End mmddyyyy-#

Where -# indicates an entry number in case multiple entries were made that day.

Searchable changes made on the following dates:

20251015-1

BROAD OUTLINE with more detail after the outline.

Everywhere in this section that I have indicated an environmental variable or EVAR, give it a name (srh did most), assign it to a global table as well as to the demographics table so that I can optionally assign specific values to individual market segments as well as to individual stocks.

Replace the a, x, y references in the broad outline below with the appropriate variable name you have assigned. Then use the REPLIT engine to incorporate these changes.

When the procedure with the analysis logic is called, first use the stock specific evvar values if any, else the market segment values if any, else the global value.

1. Accumulate 30-90 days price and major stochastic calculations history for companies with many stock shares and options being traded daily. (Start with S&P500 companies. Include Market segment representative funds. Update nightly with each successive day's data. Add other companies later.
2. Each night, identify PRIORITY stocks to watch closely during the next day's trading hours. These will be stocks with prices more than 1 standard deviation (SD) greater or less than the 21 day average price OR stocks for which we already own options.
  - a. Begin 20251015-1
  - b. End 20251015-1
  - c.
3. If STOCK\_PRICE is FALLING more after each PRICE\_CHECK\_INTERVAL (evvar) in seconds for CONSECUTIVE\_PERIODS (evvar), AND if stock price greater than 1 SD above avg OR market segment ETF average price is also falling, THEN if day trading or owning call options purchased on a previous day, sell all CALL options for that stock and use up to CASH\_INVEST\_OPTION\_PCT % (evvar) of avail cash to buy no more than OPTION\_PCT\_BUY % (evvar) of available PUT options and sell STOCK\_PCT\_SELL % (evvar) percentage of that stock owned.
4. If the stock price is RISING more after each PRICE\_CHECK\_INTERVAL (evvar) in seconds for CONSECUTIVE\_PERIODS (evvar), AND if stock price less than 1 SD below

average OR market segment ETF average price is also rising, THEN if day trading or own put options purchased on a previous day, sell all PUT options and use up to CASH\_INVEST\_OPTION\_PCT % (evar) of avail cash to buy no more than OPTION\_PCT\_BUY % (evar) of available CALL options and CASH\_INVEST\_STOCK\_PCT % (evar) of cash to buy some of that stock.

5. If the price movement decreases more than MOMENTUM\_DECR\_PCT % (evar), THEN if day trading, sell half options ELSE buy the opposite option (at the same strike price and expiry date) as was done above in sufficient quantity to lock in half our profits.
6. If the price movement reverses with increasing momentum OR loses LOSS\_PCT\_LIMIT % (evar) of the earlier profit OR we enter the final EOD\_LIMIT (evar) minutes of trading that day, THEN if day trading, sell all options for that stock, else buy more of the opposite option in sufficient quantity to avoid losing any more money.
7. Overnight repeat 1, during trading hours excl the last EOD\_LIMIT minutes, repeat 2-6.

Options purchased must be actively traded with daily volumes exceeding MIN\_OPTIONS\_VOL(environment variable evar) units traded, with a strike price being closest to OPT\_STRIKE\_PRICE\_PCT\_TARGET(evar) percent above (for CALL options) or below (for PUT options) the current stock price. The weekly expiration date will be the first Friday after tomorrow.

## MORE DETAILED DISCUSSION

Create whatever tables of data as needed, but be sure to include the following...

Table for global environment variables (perhaps here or elsewhere set a variable for paper trading or live trading for what broker and account. See references below for additional values to be set.)

Table for brokerage info including login credentials, trading fees, day trade restriction (min balance, other), day trade election yes or no . Begin with RobinHood and alpaca

Table identifying different accounts at each brokerage, name, type, total balance, cash balance

Table for stock demographics including symbol, name, sector, industry, market cap - initially populated only with data for stocks in the S&P 500 that have actively traded options available. Include an indexed smallint column called PRIORITY. After populating the stock price history table and just before the next trading session, set priority equal 1 if the closing price is greater than one standard deviation above or below the 20 day moving average, and the closing price is above or below the moving average by more than x% (percentage Target found in the environment table). Set priority to 0 for all others except the following sector ETF index funds to be included in the demographics table with a priority constantly equal 9

Sector Name, Ticker Symbol

Energy XLE

Materials XLB

Industrials XLI  
Consumer Discretionary XLY  
Consumer Staples XLP  
Healthcare XLV  
Financials XLF  
Information Technology XLK  
Communication Services XLC  
Utilities XLU  
Real Estate XLRE

Table for stock price history (populate during off hours just after the last trading session for past 90 days), preferably sending all ticker symbols and retrieving all price information from the broker in a single call rather than a separate call for each ticker symbol. Then calculate and populate into the same table common technical indicators, especially the following support for binary options: Wilder's directional movement indicators, pivot points, 14 day commodity channel index, and 20 day Bollinger bands using 2 standard deviations.

Table for Priority Current price populated with each stock ticker where priority greater than zero. record ticker symbol (Index) , datetime (Index), price at open, then new record every x seconds (from environment variables) along with percent change from previous price.

Table for Priority Archive price to which we move all records from the priority current price table at the end of each day, and purge all older than x days (set in environment table)

Table for orders showing asset, action quantity, bid, ask, limit, timedate, etc

Table for transaction log showing order ID, asset, timedate, quantity, price, LIFO gain\loss)

Support different broker APIs in separate modules (start with RobinHood and alpaca) such that if the general program decides to sell an asset, a generic call to sell can be routed to the specific sell function for the API of the active broker per the environment variables.

Evaluate the priority current price table. If no order exists for this ticker, and x periods (set in environment variables) have passed, and price has moved a greater percentage in the same direction each period then place a purchase order with less than x% (set in environment variables) of the available cash (rounded down). If all x periods exceed 2 standard deviations moving away from the moving average or all x periods are moving towards the moving average, then buy options (expiring the next Friday beyond today at the next strike price in the direction that the price is moving and that is being actively traded and The purchase represents no more than x% (set in environment variables) of the current volume of options being traded).

If the price is rising, buy call options

If the price is falling, buy put options

If we already own options, continue to monitor the current price every x seconds. When two periods have passed where the price change has been slowing down, or a period has passed where the price has not changed or has reversed, sell the options if we meet the requirements for day trading, else purchase an offsetting call or put option to lock in our current gain, selling whichever option is losing money at the next days opening, and later selling the other option when it stops gaining.

With each sale \ (purchase),

add to \ (subtract from) the brokerage account cash balance

Also, add each gain and subtract each loss from the total account balance

---

Robinhood

Alpaca

Surmount

Tradestation

Fidelity

Schwab

Original text for outline items 3 & 4 above

1.

2.

3. If market segment ETF average price is FALLING, THEN if stock price greater than average OR stock price less than 2 SD below avg AND stock price falling more and more each second, THEN sell any CALL options or use up to a% of \$\$ to buy no more than x% of available PUT options and sell b% percentage of that stock owned.

4. If market segment ETF average price is RISING, THEN if stock price less than average OR stock price greater than 2 SD above avg AND stock price rising more and more each second, THEN sell any PUT options owned for that stock or use up to a% of \$\$ to buy no more than x% of available CALL options and b% of \$\$ to buy some of that stock.

As major indices show a strong upward price trend, give more emphasis to call options.

As stock prices make new highs or bounce up from levels of support, purchase call options, but only if the next level of expected resistance represents at least a 5% price increase in the call option.

As price increases slow, stop or reverse to back test previous support levels, sell the call options, preferably as prices are still slowly increasing.

If necessary to avoid day trading, lock in the current gains by purchasing an offsetting put option, then probably sell both the next day. (Immediately sell the one losing money, then later the one gaining money)

As major indices show a strong downward price trend, give more emphasis to put options. Do the opposite for put options than what is described above for call options, but hold put options for a shorter time than we would for a call option.

---

### The Most Important Technical Indicators for Binary Options

<https://share.google/4zpXc8HUsY9JK1HRN>

#### Wilder's DMI (ADX)

Wilder's directional movement indicators and average directional index (DI) are composed of three lines, namely ADX, DI+, and DI-,

Pivot points are a technical analysis tool used by traders to determine potential support and resistance levels, that is, levels at which the price of an asset is likely to move upward or downward. They are calculated using the average of the high, low, and closing prices from earlier trading sessions.

#### Commodity Channel Index (CCI)

High values above +100 indicate the start of a solid upward trend. Values below the -100 line indicate the beginning of a strong downward trend

$$CCI = (Price - MA) / (0.015 \times D)$$

where:

Price=asset's current price

MA=moving average of asset's price

D=normal deviation from moving average

should be used with additional indicators such as oscillators.

look for divergence between the CCI and price movement, which indicates potential reversals. If the price is making new highs but the CCI fails to surpass its previous highs, it could indicate a lack of momentum and a possible reversal. look for divergence between the CCI and price movement, which indicates potential reversals. If the price is making new highs but the CCI fails to surpass its previous highs, it could indicate a lack of momentum and a possible reversal.

#### Stochastic Oscillator

The crossover of %K and %D below indicates trade entry signals. Although a 14-day period is standard, binary option traders can use the time frames they need.

$$\%K = 100 \left( \frac{C - L14}{H14 - L14} \right)$$

%D=3 period moving average of %K

where:

C=most recent closing price

L14=low of 14 previous trading sessions

H14=highest price traded during same 14-day period

Levels above 80 indicate the assets are overbought, while those below 20 indicate they are oversold.

---

## OVERVIEW

(We must do something different than a single username and password to facilitate multiple accounts working on the same project) For all accounts we need to create, I suggest that we use "RealHelp" as the username and \$oftware2Help as the pwd, unless you have a better suggestion.

Need to form private foundation rather than public charity because we are not receiving 1/3 donations from public

Need to keep the following investments in the Roth IRA before donating to the foundation to prevent paying an excise tax on jeopardizing investments.

Options

Commodities

Short sales

Probably cryptocurrency, though not listed. It may be entirely prohibited.

Foundation investments need to be in stocks and bonds, probably avoid day trading. Perhaps the two different accounts can be synchronized to support the option trades in the Roth IRA

## DATABASE OPTIONS

<https://www.lastweekinaws.com/blog/10-free-cloud-databases-you-should-consider-and-1-you-shouldnt/>

Oracle always free autonomous transaction processing. Two databases, each with 20 GB of storage. Must explicitly increase if needing more, so no surprise charges. A NoSQL database is also available.

Apache Cassandra data stax Astra DB with 40 gig storage but charges if exceeding limit. It does not support acid transactions.

Postgres NoSQL.

MySQL dynamo?

#### PROFILE

ClientID

FirstName

LastName

BirthDate

Username

Password (encrypted)

#### PHONE

ClientID

Default (Y\N)

Description

CountryCode

Number

#### ADDRESS

ClientID

Description (Mailing, Billing, Home, Shipping)

Address1

Address2

City

State

Account number

Account name

Account Description

Account type?

restrictions on

Number of trades per day

---

Help me write the SQL to create a table in a postgresql database to store all of the demographic information from Robin Hood available for stocks

To create a table in a PostgreSQL database for storing demographic information related to stocks from Robinhood, you can use the following SQL statement:

```
CREATE TABLE stock_demographics (  
  id SERIAL PRIMARY KEY,  
  symbol VARCHAR(10) NOT NULL UNIQUE,  
  name VARCHAR(255) NOT NULL,  
  sector VARCHAR(100),  
  industry VARCHAR(100),  
  market_cap BIGINT,  
  pe_ratio NUMERIC(10, 2),  
  dividend_yield NUMERIC(5, 2),  
  earnings_per_share NUMERIC(10, 2),  
  beta NUMERIC(10, 2),  
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Explanation of Columns:

id: Unique identifier for each record.

symbol: The stock ticker symbol (e.g., AAPL for Apple).

name: The name of the company.

sector: The sector in which the company operates.

industry: The specific industry of the company.

market\_cap: The market capitalization of the company in dollars.

pe\_ratio: The price-to-earnings ratio.

dividend\_yield: The dividend yield as a percentage.

earnings\_per\_share: Earnings per share figure.

beta: A measure of volatility relative to the market.



created\_at: Timestamp for when the record was created.

updated\_at: Timestamp for when the record was last updated.

Notes:

Adjust the data types and sizes based on your specific requirements.

You may want to add additional indexes or constraints depending on your query needs.

---

Help me write Python 3 code to access the Robin Hood stock API to get demographic information for each of the S&p 500 stocks and store the information in a postgresql stock\_demographics table

Sure! Below is a Python 3 script that retrieves demographic information for each S&P 500 stock from the Robinhood API and stores it in a PostgreSQL database.

Steps:

1. Use the robin\_stocks library to access Robinhood's API.
2. Retrieve stock data, focusing on demographic information.
3. Store the data in a PostgreSQL stock\_demographics table.

Note that the demographic example automatically generates the list of 500 stock symbols which can be used instead of manually entering them in the historical data example.

Prerequisites:

Install dependencies:

```
pip install robin-stocks psycpg2 pandas
```

Ensure you have a PostgreSQL database set up.

---

## Python Script

```
import robin_stocks.robinhood as rh
import psycpg2
import pandas as pd

# Robinhood Login
USERNAME = "your_robinhood_username"
PASSWORD = "your_robinhood_password"

rh.login(USERNAME, PASSWORD)

# PostgreSQL Database Credentials
DB_HOST = "localhost"
DB_NAME = "your_database"
DB_USER = "your_username"
DB_PASSWORD = "your_password"

# Connect to PostgreSQL
conn = psycpg2.connect(
    dbname=DB_NAME, user=DB_USER, password=DB_PASSWORD, host=DB_HOST
)
cursor = conn.cursor()

# Create stock_demographics table
cursor.execute("""
    CREATE TABLE IF NOT EXISTS stock_demographics (
        symbol TEXT PRIMARY KEY,
        sector TEXT,
        market_cap BIGINT,
        average_volume BIGINT,
        pe_ratio FLOAT,
        dividend_yield FLOAT
    )
""")
conn.commit()

# Get S&P 500 symbols
sp500 = pd.read_html("https://en.wikipedia.org/wiki/List_of_S%26P_500_companies")[0]
symbols = sp500["Symbol"].tolist()
```

```

# Fetch stock demographics and insert into PostgreSQL
for symbol in symbols:
    try:
        info = rh.stocks.get_fundamentals(symbol)
        if info:
            data = info[0]
            cursor.execute(
                """
                INSERT INTO stock_demographics (symbol, sector, market_cap, average_volume,
pe_ratio, dividend_yield)
                VALUES (%s, %s, %s, %s, %s, %s)
                ON CONFLICT (symbol) DO UPDATE SET
                sector = EXCLUDED.sector,
                market_cap = EXCLUDED.market_cap,
                average_volume = EXCLUDED.average_volume,
                pe_ratio = EXCLUDED.pe_ratio,
                dividend_yield = EXCLUDED.dividend_yield
                """,
                (
                    symbol,
                    data.get("sector"),
                    int(float(data.get("market_cap", 0))),
                    int(float(data.get("average_volume", 0))),
                    float(data.get("pe_ratio", 0) or 0),
                    float(data.get("dividend_yield", 0) or 0),
                ),
            )
            conn.commit()
            print(f"Inserted/Updated: {symbol}")
    except Exception as e:
        print(f"Error processing {symbol}: {e}")

# Close connections
cursor.close()
conn.close()
rh.logout()

---
```

Explanation:

1. Authenticate with Robinhood using robin\_stocks.

2. Retrieve the list of S&P 500 stocks from Wikipedia.
3. Query Robinhood API to get fundamental stock data.
4. Store the data in PostgreSQL (stock\_demographics table).
5. Handle errors & update existing records using ON CONFLICT.

---

Next Steps:

Security: Use environment variables instead of hardcoded credentials.

Enhancements: Add logging, handle rate limits, or use a different API if Robinhood's access is limited.

Would you like additional features, such as scheduled updates?

---

Help me write the SQL to create a table in a postgresql database to store all of the daily historical information from Robin Hood available for stocks

To create a table in a PostgreSQL database to store daily historical information for stocks from Robinhood, you can use the following SQL statement:

```
CREATE TABLE stock_history (  
  id SERIAL PRIMARY KEY,  
  symbol VARCHAR(10) NOT NULL,  
  date DATE NOT NULL,  
  open_price NUMERIC(10, 2),
```

```
close_price NUMERIC(10, 2),
high_price NUMERIC(10, 2),
low_price NUMERIC(10, 2),
volume BIGINT,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

#### Explanation of Columns:

id: A unique identifier for each record.

symbol: The stock ticker symbol (e.g., AAPL for Apple).

date: The date of the recorded data.

open\_price: The opening price of the stock for that day.

close\_price: The closing price of the stock for that day.

high\_price: The highest price of the stock during the day.

low\_price: The lowest price of the stock during the day.

volume: The number of shares traded that day.

created\_at: A timestamp for when the record was created.

updated\_at: A timestamp for when the record was last updated.

#### Notes:

Adjust the data types and sizes as needed based on your specific requirements.

You might want to add additional indexes or constraints based on your query needs.

---

Help me write Python 3 code to access the Robin Hood stock API to get daily historical information for the past 21 days for each of the S&p 500 stocks and store the information in a postgresql stock\_history table

To access the Robinhood API and retrieve daily historical stock data for S&P 500 stocks, you can use Python along with libraries like requests for API calls and psycopg2 or SQLAlchemy for PostgreSQL interaction. Below is an example of how you can achieve this.

Note that the demographic example automatically generates the list of 500 stock symbols which can be used instead of manually entering them in the historical data example.

## Prerequisites

1. Install the required libraries:

```
pip install requests psycopg2
```

2. Ensure you have access to the Robinhood API and have your credentials ready.

## Python Code Example

```
import requests
import psycopg2
from datetime import datetime, timedelta

# Constants
ROBINHOOD_API_URL = 'https://api.robinhood.com'
S_P500_TICKERS = ['AAPL', 'MSFT', 'GOOGL', 'AMZN'] # Add all S&P 500 tickers here
DATABASE_CONFIG = {
    'dbname': 'your_database',
    'user': 'your_user',
    'password': 'your_password',
    'host': 'localhost',
    'port': '5432'
}

# Function to get daily historical data for a stock
def get_stock_history(symbol):
    url = f'{ROBINHOOD_API_URL}/historicaldata/{symbol}/'
    params = {
        'interval': 'day',
        'span': 'week',
        'bounds': 'regular'
    }
```

```

response = requests.get(url, params=params)
if response.status_code == 200:
    return response.json()['historical']
else:
    print(f"Error fetching data for {symbol}: {response.status_code}")
    return []

# Function to store data in PostgreSQL
def store_stock_history(data, symbol):
    conn = psycopg2.connect(**DATABASE_CONFIG)
    cursor = conn.cursor()
    for entry in data:
        date = datetime.strptime(entry['begins_at'], '%Y-%m-%dT%H:%M:%S.%fZ').date()
        open_price = entry['open_price']
        close_price = entry['close_price']
        high_price = entry['high_price']
        low_price = entry['low_price']
        volume = entry['volume']

        cursor.execute("""
            INSERT INTO stock_history (symbol, date, open_price, close_price, high_price,
low_price, volume)
            VALUES (%s, %s, %s, %s, %s, %s, %s)
            ON CONFLICT (symbol, date) DO NOTHING;
            """, (symbol, date, open_price, close_price, high_price, low_price, volume))

    conn.commit()
    cursor.close()
    conn.close()

# Main function to fetch and store data for all S&P 500 stocks
def main():
    for symbol in S_P500_TICKERS:
        print(f"Fetching data for {symbol}...")
        historical_data = get_stock_history(symbol)
        if historical_data:
            store_stock_history(historical_data, symbol)

if __name__ == "__main__":
    main()

```

Notes:

1. S&P 500 Tickers: Replace S\_P500\_TICKERS with the complete list of S&P 500 tickers.

2. Database Configuration: Update DATABASE\_CONFIG with your PostgreSQL database credentials.

3. Error Handling: This code includes basic error handling. You may want to enhance it for robustness.

4. Rate Limiting: Be mindful of the API's rate limits. You might want to add a delay between requests.

5. Conflict Handling: The ON CONFLICT clause in the SQL statement prevents duplicate entries based on the symbol and date.

This code will fetch the last 21 days of historical stock data for each ticker and store it in the PostgreSQL database. Make sure you have the stock\_history table created in your database as described earlier.

---

Help me write the SQL to create a table in a postgresql database to store all of the daily historical information from Alpaca Markets available for stocks

To create a table in a PostgreSQL database for storing daily historical information for stocks from Alpaca Markets, you can use the following SQL statement:

```
CREATE TABLE alpaca_stock_history (  
  id SERIAL PRIMARY KEY,  
  symbol VARCHAR(10) NOT NULL,  
  date DATE NOT NULL,  
  open_price NUMERIC(10, 2),  
  close_price NUMERIC(10, 2),  
  high_price NUMERIC(10, 2),  
  low_price NUMERIC(10, 2),  
  volume BIGINT,  
  adjusted BOOLEAN DEFAULT TRUE,  
  dividend NUMERIC(10, 2),
```



```
split_coefficient NUMERIC(10, 2),  
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

#### Explanation of Columns:

id: Unique identifier for each record.

symbol: The stock ticker symbol (e.g., AAPL for Apple).

date: The date of the recorded data.

open\_price: The opening price of the stock for that day.

close\_price: The closing price of the stock for that day.

high\_price: The highest price of the stock during the day.

low\_price: The lowest price of the stock during the day.

volume: The number of shares traded that day.

adjusted: Indicates if the data is adjusted for splits and dividends.

dividend: The amount of dividend paid per share for that day.

split\_coefficient: The ratio of any stock splits that occurred.

created\_at: Timestamp for when the record was created.

updated\_at: Timestamp for when the record was last updated.

#### Notes:

Modify the data types and sizes as needed based on your requirements.

You may want to add additional indexes or constraints depending on your query needs.