2025-07-15 | 2:28pm ORL time

Below is a detailed, step-by-step walkthrough of the entire `trade_schwab_grok4_multi.py` script. I'll go through the code line by line or section by section, quoting the relevant code, explaining what it does, how it works, and its purpose. I'll explain everything from scratch, without assuming prior knowledge of Python or programming concepts. For example, I'll describe what an "import" is, what a "function" is, and why each line is there. The script is a complete Python program that automates stock trading decisions for three stocks (AMD, MSFT, NVDA) using data from Schwab's API and predictions from Grok 4's API. It runs in a loop, makes buy/sell decisions, logs everything to a file, and closes positions at a specific time. The code uses libraries (pre-written code from others) to handle tasks like data processing and API calls.

Section 1: Import Statements (Lines 1-11)

These lines bring in external code libraries that the script needs to use. In Python, "import" means loading code from a library so you can use its features, like tools in a toolbox. Without these, the script can't do things like handle data or make API requests.

```python
import asyncio
```

- **What it does**: Loads the entire `asyncio` library into the script.
- **How it works**: `asyncio` is a built-in Python library for "asynchronous" programming, which lets the script run tasks concurrently (like waiting for an API response while doing other things). It doesn't execute anything yet; it's just making the library available. The script uses it later for "async" functions that can pause without stopping the whole program.

```python
import pandas as pd
```

- **What it does**: Loads the `pandas` library and gives it a short name "pd" for easier use.
- **How it works**: `pandas` is an external library (installed via pip) for working with structured data, like tables or lists of numbers. "as pd" is an alias, so instead of typing "pandas" every time, you use "pd". The script uses it to create "DataFrames" (like spreadsheets) from stock price data.

```python
import numpy as np
```

- **What it does**: Loads the `numpy` library and gives it a short name "np".
- **How it works**: `numpy` is an external library for numerical computations, like handling arrays of numbers efficiently. The alias "np" shortens code. The script uses it to create and manipulate arrays of stock prices.

```python
import requests
```

- **What it does**: Loads the `requests` library.
- **How it works**: `requests` is an external library for making HTTP requests (like sending data to a website or API and getting a response back). The script uses it to send requests to the Grok 4 API for predictions.

```python
import logging
```

- **What it does**: Loads the `logging` library.
- **How it works**: `logging` is a built-in Python library for recording messages (like errors or status updates) to a file or console. The script uses it to write all events to a log file instead of printing them, which is useful for tracking what happens without watching the screen.

```python
import re
```

- **What it does**: Loads the `re` library.
- **How it works**: `re` is a built-in library for "regular expressions," a way to search for patterns in text (e.g., finding specific formats in log lines). The script uses it to parse trade entries from the log file.

```python
from datetime import datetime, time, timedelta, timezone
```

- **What it does**: Loads specific tools (classes) from the built-in `datetime` library: `datetime` for handling dates/times, `time` for time objects, `timedelta` for time differences, `timezone` for timezone info.
- **How it works**: "from ... import ..." loads only specific parts. The script uses these to get the current time, check if it's 3:58 PM ET, and calculate intervals (e.g., subtract 4 hours for ET from UTC).

```python
from schwab.auth import easy_client
```

- **What it does**: Loads the `easy_client` function from the `schwab.auth` module in the `schwab-py` library.
- **How it works**: `schwab-py` is an external library (installed via pip) for interacting with Schwab's API. This function simplifies setting up the connection. The script uses it to create a client for API calls.

```python
from schwab.orders.equities import equity_buy_market, equity_sell_marke
```

- **What it does**: Loads two functions from the `schwab.orders.equities` module in `schwab-py`: `equity_buy_market` for buy orders at market price,

`equity_sell_market` for sell orders.

- **How it works**: These functions create order objects for stock trades. The script uses them to place simulated buy/sell orders.

```python
from sklearn.preprocessing import MinMaxScaler
```

- **What it does**: Loads the `MinMaxScaler` class from the `sklearn.preprocessing` module in scikit-learn.
- **How it works**: scikit-learn is an external library for machine learning tools. `MinMaxScaler` normalizes data to a 0-1 range. The script uses it to scale stock prices before sending to Grok 4.

Section 2: Logging Configuration (Line 13)

```python
logging.basicConfig(filename='/home/gmafanasiev/trading.log', level=log
```

- **What it does**: Sets up the logging system to write messages to a file.
- **How it works**: `basicConfig` is a function from the `logging` library. `filename` specifies the file path where logs are saved (`/home/gmafanasiev/trading.log` is your PythonAnywhere user directory). `level=logging.INFO` means it logs "INFO" level messages and higher (e.g., errors). `format` adds a timestamp (`%(asctime)s`) and the message (`%(message)s`) to each log entry. From now on, any `logging.info("message")` writes to this file.

Section 3: API Credentials (Lines 16-20)

```python
SCHWAB_APP_KEY = "5AqBefcOK5NJbuv0twAo6s41Ulc2cRlr"
SCHWAB_APP_SECRET = "o76s0GYzvACjHO1e"
SCHWAB_CALLBACK_URL = "https://127.0.0.1:8182"
XAI_API_KEY = "xai-BU2X5OnnITYxBZN5Q3dc8mNrbyw4RrxdrM7caEZvaDZJpASF0e8v
TOKEN_PATH = "/home/gmafanasiev/token.json"
```

- **What it does**: Defines variables (like named storage boxes) for API keys and paths.
- **How it works**: These are strings (text values in quotes) assigned to variables. `SCHWAB_APP_KEY` and `SCHWAB_APP_SECRET` are from Schwab Developer Portal for authentication. `SCHWAB_CALLBACK_URL` is the redirect URL for OAuth. `XAI_API_KEY` is from xAI for Grok 4. `TOKEN_PATH` is the file where the authentication token saves. These are used later in the client setup and API calls.

Section 4: Trading Parameters (Lines 23-32)

```python
SYMBOLS = ["AMD", "MSFT", "NVDA"]
SEQUENCE_LENGTH = 60
RISK_PER_TRADE = 0.01  # Initial 1% risk, divided across stocks
MAX_EQUITY = 1000  # Maximum equity assumption per trade
STOP_LOSS_PCT = 0.02
TAKE_PROFIT_PCT = 0.04
TIMEFRAME = "DAY"
UPPER_THRESHOLD = 0.65  # Buy if > this (no-trade buffer upper)
LOWER_THRESHOLD = 0.35  # Sell if < this (no-trade buffer lower)
ADJUSTMENT_INTERVAL = 30 * 60  # Adjust every 30 minutes
```

- **What it does**: Defines constants (fixed values) for trading rules.
- **How it works**: `SYMBOLS` is a list (array) of stock codes to trade. `SEQUENCE_LENGTH` is an integer (60) for days of data. `RISK_PER_TRADE` is a float (0.01 = 1%) for risk calculation. `MAX_EQUITY` is an integer (1000) to cap risk amount. `STOP_LOSS_PCT` is 0.02 (2%) for loss limit. `TAKE_PROFIT_PCT` is 0.04 (4%) for profit target. `TIMEFRAME` is a string ("DAY") for data type. `UPPER_THRESHOLD` is 0.65 for buy decisions. `LOWER_THRESHOLD` is 0.35 for sell decisions. `ADJUSTMENT_INTERVAL` is 1800 (30*60) seconds for reviews. Comments (#) are notes, ignored by Python.

Section 5: Schwab Client Initialization (Lines 35-40)

```python
client = easy_client(
    api_key=SCHWAB_APP_KEY,
    app_secret=SCHWAB_APP_SECRET,
    callback_url=SCHWAB_CALLBACK_URL,
```

```
    token_path=TOKEN_PATH
)
```

- **What it does**: Creates an object (client) for interacting with Schwab's API.
- **How it works**: Calls `easy_client` function with your key, secret, callback URL, and token path. This handles OAuth authentication: if no token, prints a URL for browser login; saves token to file. The `client` variable is used for all Schwab calls (e.g., get prices).

Section 6: `fetch_bar_data` Function (Lines 42-51)

python

```
async def fetch_bar_data(symbol, timeframe="DAY", limit=60):
    resp = client.get_price_history_every_day(symbol, limit=limit)
    if resp.status_code == 200:
        data = resp.json()
        prices = [candle["close"] for candle in data["candles"]]
        df = pd.DataFrame(prices, columns=["close"])
        return df
    else:
        logging.info(f"Failed to fetch data for {symbol}: {resp.status_
        return pd.DataFrame()
```

- **What it does**: Defines a function (reusable code block) to fetch historical stock prices.
- **How it works**: `async def` makes it asynchronous (can pause). Takes parameters: symbol (e.g., "AMD"), timeframe (default "DAY"), limit (default 60). Calls Schwab API via `client` to get price history. If status 200 (success), parses JSON response, extracts "close" prices from "candles" list, creates Pandas DataFrame with "close" column, returns it. Else, logs error code, returns empty DataFrame.

Section 7: `prepare_grok4_input` Function (Lines 53-61)

python

```
def prepare_grok4_input(data, sequence_length):
    prices = data['close'].values.reshape(-1, 1)
    scaler = MinMaxScaler()
    scaled_prices = scaler.fit_transform(prices)
```

```
    sequences = []
    for i in range(len(scaled_prices) - sequence_length):
        sequences.append(scaled_prices[i:i + sequence_length])
    return np.array(sequences)
```

- **What it does**: Defines a function to prepare data for Grok 4.
- **How it works**: Takes DataFrame `data` and `sequence_length` (60). Extracts 'close' column as array, reshapes to 2D (rows x 1 column). Creates scaler object, fits to data and transforms to 0-1 range. Creates empty list. Loops from 0 to length minus 60, appending 60-row slices. Converts list to NumPy array, returns it.

Section 8: `get_grok4_prediction` Function (Lines 63-74)

python

```
def get_grok4_prediction(data, symbol):
    headers = {"Authorization": f"Bearer {XAI_API_KEY}"}
    data_str = ",".join(map(str, data.flatten()))
    payload = {
        "prompt": f"Predict {symbol} price direction (0 for down, 1 for
        "model": "grok-4"
    }
    response = requests.post("https://api.x.ai/v1/predict", json=payloa
    return float(response.json()["prediction"])
```

- **What it does**: Defines a function to get a prediction from Grok 4.
- **How it works**: Takes data array and symbol. Sets headers with API key (Bearer token for authentication). Flattens data to 1D, converts to strings, joins with commas. Creates dictionary `payload` with prompt (instructions to Grok) and model name. Sends POST request to xAI API with JSON payload and headers. Gets response, parses JSON, extracts "prediction" key as float, returns it (0 to 1 value).

Section 9: `analyze_trades` Function (Lines 76-124)

This is a long function; I'll explain in parts.

python

```
def analyze_trades(log_file_path):
    try:
        with open(log_file_path, 'r') as f:
```

```
        with open(log_file_path, 'r') as f:
            log_content = f.read()
```

- **What it does**: Opens and reads the log file.
- **How it works**: `try` block handles errors. `with open` opens file in read mode ('r'), assigns to `f`. `f.read()` reads all text into `log_content`.

python

```
trades = []
        buy_pattern = r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) - Placed
        sell_pattern = r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) - Placed
```

- **What it does**: Sets up an empty list for trades and defines regex patterns.
- **How it works**: `trades = []` creates empty list. `buy_pattern` is a string with regex to match buy log lines (captures timestamp, type, qty, symbol, price). `sell_pattern` does the same for sells. `r''` denotes raw string for backslashes.

python

```
today = pd.Timestamp.now().strftime('%Y-%m-%d')
        for line in log_content.split('\n'):
            buy_match = re.search(buy_pattern, line)
            if buy_match and buy_match.group(1).startswith(today):
                trades.append({
                    'timestamp': buy_match.group(1),
                    'type': 'BUY',
                    'trade_type': buy_match.group(2),
                    'symbol': buy_match.group(4),
                    'qty': int(buy_match.group(3)),
                    'price': float(buy_match.group(5))
                })
            sell_match = re.search(sell_pattern, line)
            if sell_match and sell_match.group(1).startswith(today):
                trades.append({
                    'timestamp': sell_match.group(1),
                    'type': 'SELL',
                    'trade_type': sell_match.group(2),
                    'symbol': sell_match.group(4),
                    'qty': int(sell_match.group(3)),
                    'price': float(sell_match.group(5))
                })
```

- **What it does**: Gets today's date and parses log lines for today's trades.
- **How it works**: `pd.Timestamp.now()` gets current time as Pandas timestamp, `.strftime('%Y-%m-%d')` formats to YYYY-MM-DD. Loops over each line (split by newline). Uses `re.search` to match patterns. If match and timestamp starts with today, extracts groups (captured parts) and appends dictionary to `trades` list. Converts qty to int, price to float.

python

```
pl_by_stock = {symbol: {'long': 0, 'short': 0} for symbol in SYMBOLS}
        for i in range(1, len(trades)):
            prev_trade = trades[i-1]
            curr_trade = trades[i]
            if prev_trade['symbol'] == curr_trade['symbol']:
                qty = min(prev_trade['qty'], curr_trade['qty'])
                if prev_trade['type'] == 'BUY' and prev_trade['trade_ty
                    pl = (curr_trade['price'] - prev_trade['price']) *
                    pl_by_stock[prev_trade['symbol']]['long'] += pl
                elif prev_trade['type'] == 'SELL' and prev_trade['trade
                    pl = (prev_trade['price'] - curr_trade['price']) *
                    pl_by_stock[prev_trade['symbol']]['short'] += pl
```

- **What it does**: Calculates profit/loss (P/L) for each stock's long/short trades.
- **How it works**: Creates dictionary with symbols as keys, each with 'long' and 'short' at 0. Loops over trades starting from second (range 1 to len). Compares consecutive trades: if same symbol, calculates qty as min of both. If long open-close, P/L = (sell price - buy price) * qty, adds to long. If short open-close, P/L = (sell price - buy price) * qty (profit if price dropped), adds to short.

python

```
total_pl = sum(sum(pl.values()) for pl in pl_by_stock.values())
        total_trades = len(trades)
        win_rate = sum(1 for pl in pl_by_stock.values() for v in pl.val

        return {'pl': total_pl, 'trades': total_trades, 'win_rate': win
    except Exception as e:
        logging.info(f"Error: {str(e)}")
        return {'pl': 0, 'trades': 0, 'win_rate': 0, 'pl_by_stock': {sy
```

- **What it does**: Computes totals and returns summary.
- **How it works**: `total_pl` sums all P/L values. `total_trades` is trades list length. `win_rate` counts positive P/L entries, divides by trades (max 1 to avoid division by zero). Returns dictionary with totals. `except` catches errors, logs, returns zeros.

Section 10: `get_grok4_adjustments` Function (Lines 126-141)

python

```
def get_grok4_adjustments(trade_summary):
    headers = {"Authorization": f"Bearer {XAI_API_KEY}"}
    prompt = f"Analyze today's trading: Total P/L: ${trade_summary['pl'
    payload = {
        "prompt": prompt,
        "model": "grok-4"
    }
    response = requests.post("https://api.x.ai/v1/predict", json=payloa
    try:
        result = response.json()
        threshold = min(max(float(result.get("threshold", 0.5)), 0.45),
        risk = min(max(float(result.get("risk", 0.01)), 0.005), 0.015)
        return threshold, risk
    except:
        return 0.5, 0.01
```

- **What it does**: Sends trade summary to Grok 4 for adjustment suggestions.
- **How it works**: Sets headers with API key. Builds prompt string with formatted P/L, trades, win rate. Creates payload dictionary. Sends POST to xAI API. Parses response JSON, extracts "threshold" and "risk" (defaults if missing), clamps to ranges, returns them. If error, returns defaults.

Section 11: `get_position` Function (Lines 143-152)

```python
def get_position(account_hash, symbol):
    try:
        resp = client.get_account(account_hash)
        if resp.status_code == 200:
            positions = resp.json()["positions"]
            for pos in positions:
                if pos["instrument"]["symbol"] == symbol:
                    return float(pos["quantity"])
        return 0
    except:
        return 0
```

- **What it does**: Gets current share quantity for a stock.
- **How it works**: Calls Schwab API for account positions. If success, parses JSON, loops through positions list, matches symbol, returns quantity as float. Returns 0 if not found or error.

Section 12: `trading_logic` Function (Lines 154-245)

The core loop.

```python
async def trading_logic():
    # Get account hash
    resp = client.get_account_numbers()
    if resp.status_code != 200:
        logging.info(f"Failed to get account hash: {resp.status_code}")
        return
    account_hash = resp.json()[0]["hashValue"]
    logging.info(f"Account Hash: {account_hash}")
```

- **What it does**: Fetches account identifier.
- **How it works**: Calls API for accounts, exits if fail. Extracts hash from first account, logs it.

```python
last_adjustment_time = 0
    while True:
        try:
            # Fetch account info
            resp = client.get_account(account_hash)
            if resp.status_code != 200:
                logging.info(f"Failed to get account info: {resp.status
                await asyncio.sleep(60)
                continue
            account = resp.json()
            equity = float(account["accountValue"])
            logging.info(f"Account Equity: ${equity:.2f}")
```

- **What it does**: Infinite loop to fetch equity periodically.
- **How it works**: Sets timer to 0. `while True` loops forever. `try` catches errors. Calls API for account, skips 60 seconds if fail. Parses JSON, extracts equity as float, logs formatted.

```python
# Close positions at 3:58 PM ET
            current_time = datetime.now(timezone.utc) - timedelta(hours
            if current_time.time() >= time(15, 58):
                for symbol in SYMBOLS:
                    position_qty = get_position(account_hash, symbol)
                    if position_qty > 0:
                        order = equity_sell_market(symbol, position_qty
                        client.place_order(account_hash, order, dry_run
                        logging.info(f"Closed long position for {symbol
                    elif position_qty < 0:
                        order = equity_buy_market(symbol, abs(position_
                        client.place_order(account_hash, order, dry_run
                        logging.info(f"Closed short position for {symbo
                await asyncio.sleep(60)
                continue
```

- **What it does**: Closes all positions if time is 3:58 PM ET or later.
- **How it works**: Gets current ET time. If >= 3:58 PM, loops symbols, gets qty, sells if positive (long), buys if negative (short), using market orders (dry_run simulates).

Logs closures, sleeps 60 seconds, skips to next loop.

```python
# Analyze trades every 30 minutes
            current_time_ts = current_time.timestamp()
            if current_time_ts - last_adjustment_time >= ADJUSTMENT_INT
                trade_summary = analyze_trades('/home/gmafanasiev/tradi
                threshold, risk_per_trade = get_grok4_adjustments(trade
                logging.info(f"Grok 4 Adjustments - Threshold: {thresho
                last_adjustment_time = current_time_ts
            else:
                threshold, risk_per_trade = PREDICTION_THRESHOLD, RISK_
            risk_per_stock = risk_per_trade / len(SYMBOLS)
```

- **What it does**: Checks and adjusts parameters every 30 minutes.
- **How it works**: Gets timestamp. If time passed >= 1800 seconds, calls `analyze_trades` on log, gets adjustments from Grok, logs them, updates timer. Else, uses initial values. Calculates per-stock risk (e.g., 0.01 / 3).

```python
# Process each stock
            for symbol in SYMBOLS:
                bars = await fetch_bar_data(symbol, TIMEFRAME, SEQUENCE
                if bars.empty:
                    logging.info(f" No data retrieved for {symbol}. Ret
                    continue

                sequences = prepare_grok4_input(bars, SEQUENCE_LENGTH)
                if len(sequences) == 0:
                    logging.info(f"Insufficient data for Grok 4 for {sy
                    continue

                latest_sequence = sequences[-1]
                prediction = get_grok4_prediction(latest_sequence, symb
                current_price = bars.iloc[-1]['close']
                position_qty = get_position(account_hash, symbol)
                logging.info(f"{symbol} - Current Price: ${current_pric

                if prediction > UPPER_THRESHOLD and position_qty == 0:
                    risk_amount = min(equity * risk_per_stock, MAX_EQUI
                    stop_loss_price = current_price * (1 - STOP_LOSS_PC
```

```
                qty = int(risk_amount / (current_price - stop_loss_
                if qty > 0:
                    order = equity_buy_market(symbol, qty)
                    messages, success = client.place_order(account_
                    if success:
                        logging.info(f"Placed BUY order (dry run, l
                    else:
                        logging.info(f"Buy order failed for {symbol

            elif prediction < LOWER_THRESHOLD and position_qty == 0
                risk_amount = min(equity * risk_per_stock, MAX_EQUI
                stop_loss_price = current_price * (1 + STOP_LOSS_PC
                qty = int(risk_amount / (stop_loss_price - current_
                if qty > 0:
                    order = equity_sell_market(symbol, qty)
                    messages, success = client.place_order(account_
                    if success:
                        logging.info(f"Placed SELL order (dry run,
                    else:
                        logging.info(f"Sell order failed for {symbo

            elif prediction < LOWER_THRESHOLD and position_qty > 0:
                order = equity_sell_market(symbol, position_qty)
                messages, success = client.place_order(account_hash
                if success:
                    logging.info(f"Placed SELL order (dry run, clos
                else:
                    logging.info(f"Sell order failed for {symbol}:

            elif prediction > UPPER_THRESHOLD and position_qty < 0:
                order = equity_buy_market(symbol, abs(position_qty)
                messages, success = client.place_order(account_hash
                if success:
                    logging.info(f"Placed BUY order (dry run, close
                else:
                    logging.info(f"Buy order failed for {symbol}: {
```

- **What it does**: Loops over each stock to fetch data, predict, and trade.
- **How it works**: For each symbol: Fetches bars asynchronously. If empty, logs and skips. Prepares sequences. If none, skips. Takes last sequence, gets prediction. Gets current close price from DataFrame. Gets position qty. Logs price/prediction/position. Checks prediction: - > UPPER (0.65), no position:

Calculates risk (min of % and MAX_EQUITY), stop-loss, qty (shares to buy based on risk / stop-loss distance), places buy order if qty >0, logs. - < LOWER (0.35), no position: Similar for short sell. - < LOWER, positive position: Sells to close long. - > UPPER, negative position: Buys to close short. Uses `dry_run=True` to simulate.

```python
except Exception as e:
            logging.info(f"Error: {str(e)}")

        await asyncio.sleep(60)
```

- **What it does**: Catches any errors in the loop, logs them, then pauses 60 seconds.
- **How it works**: `except` logs error message. `await asyncio.sleep(60)` pauses asynchronously for 1 minute before next loop.

Section 13: `main` Function (Lines 247-249)

```python
async def main():
    logging.info(f"Starting AI-powered day trader for {SYMBOLS} with Gr
    await trading_logic()
```

- **What it does**: Defines the entry function.
- **How it works**: Logs start message with symbols. Calls and waits for `trading_logic`.

Section 14: Execution Block (Lines 251-252)

```python
if __name__ == "__main__":
    asyncio.run(main())
```

- **What it does**: Runs the script if executed directly.
- **How it works**: `if __name__ == "__main__"` checks if the file is run as a script (not imported). `asyncio.run(main())` starts the asynchronous `main` function,

launching the loop.