* Introduction
  + Algorithmic trading explanation
    - Strategy based on how the market behaves
    - Strategy example
      * Look over avg of last 30 days and if price goes below this average, we buy some stock. If it goes above, we sell.
  + Will be a lot of trading terminology
* Motivation
  + Optiver Traders : need for a autotrading testing platform to prototype algorithms without being exposed to monetary loss.
  + Idea: Create a platform where we can simulate the execution of orders. See how strategy performs but without spending real money on the stock market
    - As all the traders only familiar with Python, that’s the language chosen
* Current projects
  + Each have many downfalls
    - Very difficult to learn
    - Costly to run
    - Poor / outdated documentation
    - No visual metrics to measure performance
    - Deeply intertwined with a specific market data API
  + Issue that all have
    - Impossible to test out multiple strategies
* My solution
  + Fixes above issues
    - Syntax cohesive to non-programmers & no advanced threading knowledge needed
    - Free at least to start
    - Automated documentation pipeline
    - Live graphing tool to compare and contrast performance
    - API agnostic.
  + Notably, allows for parallel execution of multiple strategies.
    - No other platform allows for this
    - This allows comparison of how different strategies perform over the same market data
    - Through hyperparameter tuning, its easy to experiment with comparing variations of the same strategy.
      * Most interesting part for the traders who have tried out the platform so far
* Difficulty of multiple strategy parallelisation hasn’t been attempted before
  + Algorithmic strategies can be quite CPU intensive and can’t have a high latency
  + Logical solution would be to run framework in a single process and dedicate each registered user strategy its own thread.
  + Python GIL prevents this
    - The Global Interpreter can only be held by one thread at once.
    - Whilst Python multithreading appears to run in parallel, thread execution is actually interleaved. So a Python multithreading application runs concurrently but not in parallel, leading to great performance losses.
* I had the idea of separating each distinct user strategy into a separate processes to solve this issue. All of these strategy processes, which are in a process pool, are controlled by and monitored by a central master process. Whilst inter-process communication over shared memory blocks can hamper performance, the final system design and the synchronisation algorithms mean this doesn’t degrade the execution of strategies.
* System Design
  + Provide overall system design
    - Markets & SIP feed
    - 3rd Party API
      * Streamed over Websocket
    - Python App
      * User strategy codes
      * Central Store (SHM)
      * Streaming data component
      * Subscription Component
      * Component to simulate execution
      * Graphing Component
  + Focus in on Python App
  + Have boxes covering more detailed system design
  + USER STRATEGY
    - Want to retrieve the latest prices for user data from the central store
    - After interpreting this data, want to place orders to be executed virtually.
  + CENTRAL STORE
    - Hold a central store, basically a hashmap, of each stock, and the latest price of that strategy.
  + STREAMING DATA
    - Takes in data via websocket
    - API agnostic component that is specific to the market data API used
      * Very easy to switch out
    - Takes in latest prices and places in the central repository
  + Already there are issues here. We have a several strategies wanting to read from the central store and one component wishing to write into the store. Good example of the synchronicity problems that I encountered.
    - Classic example of readers and writers problem.
      * Typical example used in lectures has writer starvation
      * Read papers to see what algorithms would be best suited
      * As only a single writer, I realised I could use a modification of Ballhausen’s algorithm, which only requires a simple semaphore.
      * General idea is that one writer takes up the same space as all readers together. This means that for a writer to write, it must wait on the semaphore N times, where N is the number of readers. This ensures readers do not enter their critical section when writing occurs.
      * Ensures progress with no reader nor writer starvation.
  + SUBSCRIPTIONS
    - Relatively simple component that keeps track of which strategies are subscribed to which data
    - When a user strategy wishes to subscribe to data from a particular stock symbol, it places a SubscriptionEvent object in a synchronised queue in shared memory.
    - Notably these events can be subscribe or unsubscribe
    - The subscription manager reads these events and subscribes to
    - Ensures we only receive to the stocks that at least one strategy
  + EXECUTION SIMULATION
    - One component per strategy
    - User code calls functions on this object to place orders
    - Based on the current stock price retrieved from the central store, simulates the execution of open orders
    - Interesting algorithms behind fast execution of open orders
      * Most orders have a limit price. So if we place a bid order with a limit of 150, it won’t be executed until the stock goes below 150.
      * Naïve way to execute orders is to loop through every order that is current placed and check if the order can be executed.
        + Not reasonable when each strategy can submit thousands of orders across many stocks
      * For each stock the strategy has placed orders for maintains a Dual Heap of buy and sell orders
        + One max heap to keep track of the bid orders
        + One min heap to keep track of ask orders
        + Semi-ordering of the heap based on the limit prices specified.
      * For bid orders, we can poll max heap and check whether the ask price on the real exchange is less than top of the heap.
      * For ask orders, we can poll min heap and check whether the real exchange bid price is more than the top of the heap.
  + ERROR MANAGEMENT
    - Currently, there is no way for the master process to know the status of the strategy processes.
    - If one encounters an error, it has no idea.
    - Additionally, there is no output to the user, as the error has occurred in a separate process.
    - When a strategy encounters an error, it places the error object onto a synchronised queue in shared memory.
    - Then we have a thread in main memory that waits for a error object to be placed in the queue.
    - Once the main processes receives the error object, the main process will rethrow the error if required, meaning that the stack trace will be output to the user. Then the main process can safely join the other strategy processes that didn’t error.
* Writing to disk & graphing
  + Adding a visualisation tool that takes in data from all strategies
  + We create yet another process that hosts a Dash server, which allows the user to create their own graphs
  + Very useful for comparing the profit and loss of each strategy
  + You can also see how the positions for each stock in each strategy vary over time.
  + User can easily create their own, more complex graphs using matplotlib (DO ONE WITH PRICE AND POSITIONS OVER TIME).
  + To get the data for the graphs, one can either call the necessary functions in the exchange object to retrieve easy to use dataframe objects. Otherwise, all transactions and positions are saved to disk for each run. So a user can easily step through execution of their strategy line by line.
* Documentation Pipeline
  + As mentioned, all the other libraries suffer immensely from poor documentation
  + So to ensure this doesn’t happen, I’ve put a lot of work into the publishing and documentation pipeline
    - After building a new version, it is published immediately to the PIP library for others to use
    - At the same time, I use sphinx to automatically document the entire module. This effectively scrapes all the python files, reading the docstrings and creating html website ready to be uploaded
    - These html files are committed to github where github Actions pushes them to github pages, which is where the documentation site is hosted.
    - Show docs pics.
* Issues Faced
  + Pickling
  + Market hours restricting development time
    - Decided to create an mock API that stochastically mimics the movement in price for every stock subscribed to.
      * Good example of API agnostic nature of the market data component.
      * Just took an hour to create and replace the Alpaca Data Streamer
      * Allows for development outside market hours too
* Example usage
  + Given the framework to various quantitative researchers
  + Here is a simple strategy coded by Matthew Verasamy:
    - Mean reversion strategy DEMO
      * Grid search over parameters