**Ib\_insync Tutorial**

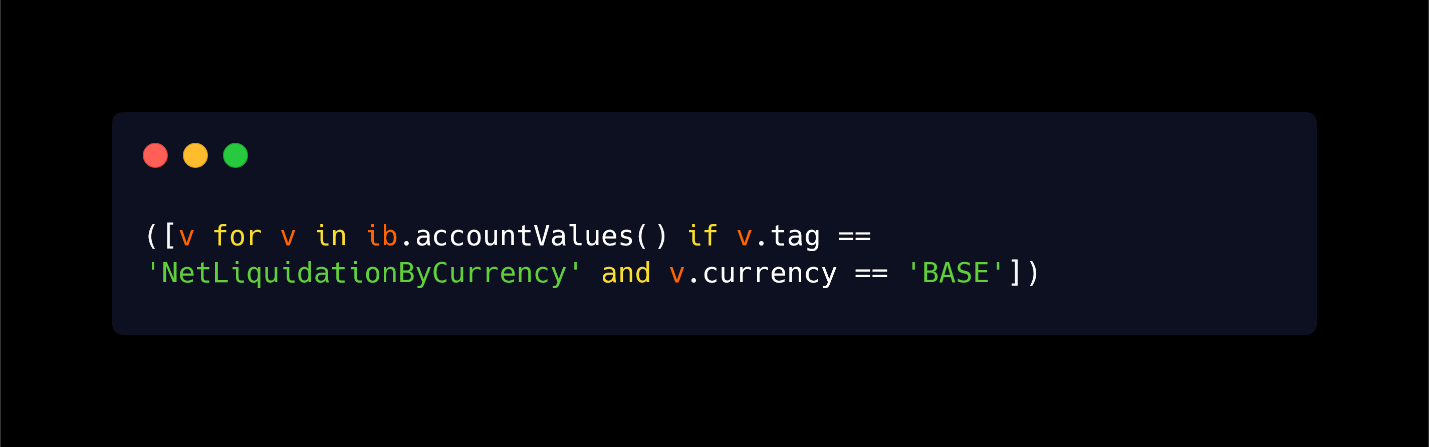
* **Overview:**
  + Provides both a blocking and an asynchronous interface to the IB API, using asyncio networking and event loop.
  + The IB class offers **direct access to the current state**, such as orders, executions, positions, tickers etc. This state is automatically kept in sync with the TWS/IBG application.
  + Blocking: Will block until complete and return the result. The current state will be kept updated while the request is ongoing.
  + Asynchronous: All methods that have the “Async” postfix. Implemented as coroutines or methods that return a Future
  + **The One Rule:** The one rule when working with the IB class is that **user code may not block for too long**.
    - While some of the request methods are blocking, the framework will keep spinning in the background and handle all messages received from TWS/IBG
    - It is important to not block the framework from doing its work. If, for example, the user code spends much time in a calculation, or uses time.sleep() with a long delay, the framework will stop spinning, messages accumulate, and things may go wrong
    - IB request methods are okay to use and do not count towards the user operation time, no matter how long the request takes to finish.
    - An example of too long is if the timestamp of tick data is to remain accurate within a millisecond**, then the user code must not spend longer than a millisecond**. If, on the other extreme, there is very little incoming data and there is no desire for accurate timestamps, **then the user code can block for hours.**
    - If a user operation takes a long time then it can be farmed out to a different process. Alternatively the operation can be made such that it periodically calls IB.sleep(0)
    - For introducing a delay, never use time.sleep() but use [sleep()](https://ib-insync.readthedocs.io/api.html#ib_insync.ib.IB.sleep) instead.
* **Basics:**
  + **Installation:** Use pip install ib\_insync, whilst ensuring that the client TWS/IB is installed and the API is enabled
  + **Imports:** We can import the library using from ib\_insync import \*
  + **Connection:** We can instantiate an IB class and use the .connect() method on it**.** The method is a blocking method.

**Text, logo

Description automatically generated**

* + **Current State:** If the connection succeeded, then ib will be synchronized with TWS/IBG. The “current state” is now available via methods such as ib.positions(), ib.trades(), ib.openTrades(), ib.accountValues() or ib.tickers(). An example of current state is that we can filter account values to get the liquidation value

**Text

Description automatically generated**

* + **Contract:** Contracts can be specified in different ways:
    - The ibapi way in which we create an empty Contract object and set its attributes one by one
    - By using Contract and giving the attributes as keyword argument
    - By using the specialized Stock, Option, Forex, Index, CFD, Commodity, Bond, FuturesOption, MutualFund, Warrant
    - The following are some examples:

**Text

Description automatically generated**

* + **Sending a Request:** The IB class has nearly all request methods that the IB API offers. The methods that return a result will block until finished and then return the result
    - **Current State vs Request:** Doing a request involves network traffic going up and down and can take considerable time. The current state however is always immediately available. So it is preferable to use the current state methods over requests. For example use ib.openOrders() in preference over ib.reqOpenOrders(), or ib.positions() over ib.reqPositions()
  + **Logging:** The following will put log messages of information

Graphical user interface, text, application

Description automatically generated

* + **Disconnect:** The following will disconnect ib and clear all its state

Graphical user interface, text, application

Description automatically generated

* **Contract Details:**
  + Suppose we want to find contract details for APPLE stock
    - Start by creating a Stock object and request details for it using reqContractDetails(stock\_object)

Text

Description automatically generated



* + - **Text

      Description automatically generated**We get a long list of contract details, we can print the first one using cds[0]
* Graphical user interface, application

  Description automatically generatedThe contract itself is int the contract property of the contract details, we can make list of contracts, convert it to a Dataframe using a utility function
* We can see that the AAPL stock trades on different exchanges in different currencies. If we want the one in USD on the SMART exchange, we can adjust the Stock object we created

Graphical user interface, text, application

Description automatically generated

* We try a non-existing contract as well

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

* With qualifyContracts method the extra information that is sent back from the contract details request is filled in the original contracts

Graphical user interface, text

Description automatically generated

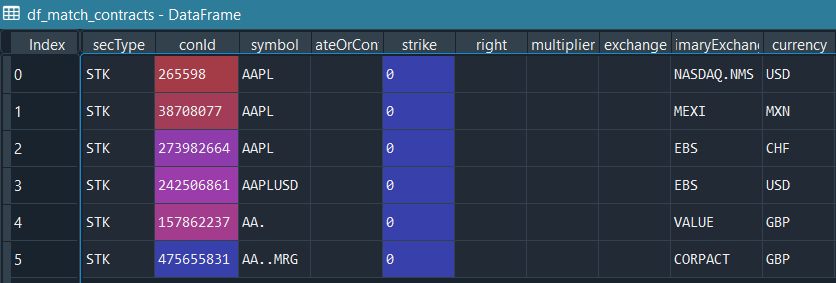
Text

Description automatically generated

* There is also an API function to request stock that matches a pattern, for example we want stocks that have the symbol AAPL

Text

Description automatically generated



* **Bar Data:**
  + **Historical Data:** To get the earliest date of available bar data, the head timestamp can be requested

****

* + **Hourly Data over a Timeframe:** To request hourly data over the last 60 trading days we can do the following:

**Text

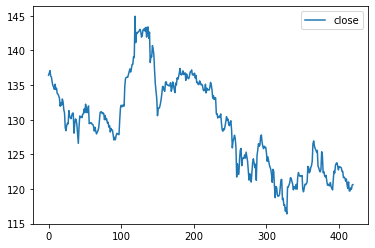
Description automatically generated**

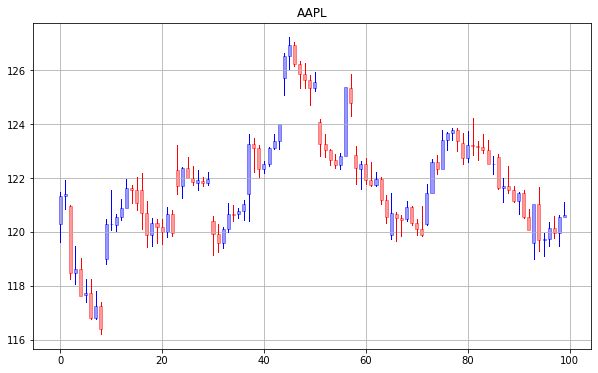


* + **Plotting Data:** We can plot data as different graphs, in the following examples we plot them as a line and candlestick graph
    - We want to plot a line graph for the closing price of the Apple Stock, we can do the following

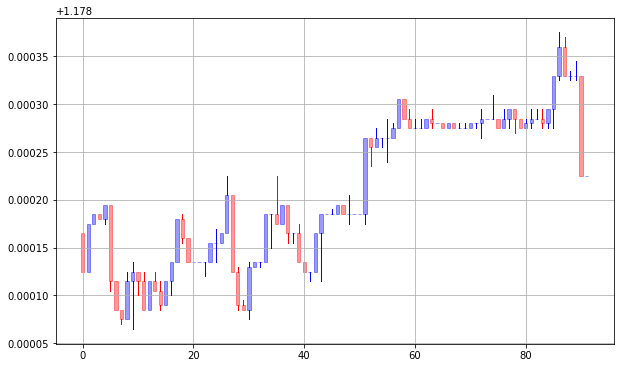
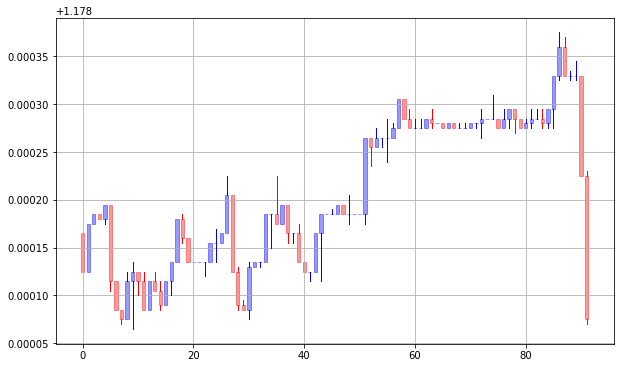
**Graphical user interface, text, application

Description automatically generated**



* + - There is also a utility function to plot bars as a candlestick plot. It can accept either a DataFrame or a list of bars. Here it will print the last 100 bars:
  + Text

    Description automatically generated**Historical Data with Real-time Updates:** We can get live updates for historical bars. This is done by setting endDateTime to an empty string and the keepUpToDate parameter to True.



* + **Real Time Bars:** With reqRealTimeBars a subscription is started that send a new bar every 5 second
    - Start by setting up an event handler for bar updates, in our case we will print the open price every 5 seconds, over half a minute
    - We will then do the real request and connect the event handler
    - We will let it run for half a minute and then cancel the real time bars

**Text

Description automatically generated**

Text

Description automatically generated

* **Tick Data:**
  + **Streaming Tick Data:** 
    - We can start by creating some contracts, in our case we will create Forex in different currencies and qualify them. The following image shows one of these contracts

****

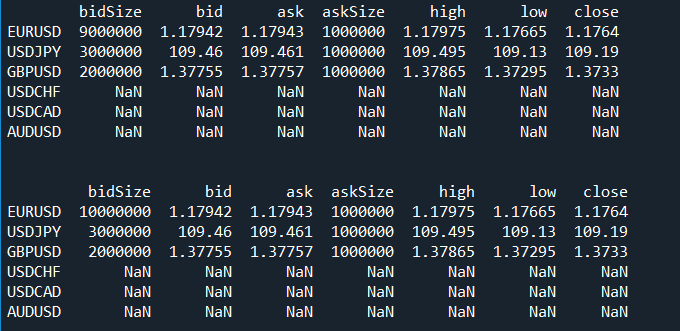
Text

Description automatically generated

* + - Next, we can create a 0.5 second loop, that prints a live updated ticker table, it is updated on every ticker change
    - New tick data is available in the “ticks” attribute of the pending tickers. The tick data will be cleared before the next update

**Text

Description automatically generated**



* + **Tick by Tick Data:**
    - The ticks above are time-sampled by IB, with reqMktData, not every tick from the exchange is send. To obtain every tick, just as it appears on TWS, we can use reqTickByTickData, however this is limited to only three simultaneous subscriptions, where bid-ask ticks and sale ticks take up a subscription each
  + **Historical Tick Data:**
    - **Text

      Description automatically generatedText

      Description automatically generated**Historical tick data can be fetched with a maximum of 1000 ticks at a time, either the start or end time must be given and one of them must remain empty.
* **Option Chains:**
  + Suppose we want to find options on the SPX, with the following conditions: use the next three-monthly expiries, use strike prices within +- $20 of the current SPX value, use strike prices that are a multitude of $5
    - To get the current market value, we first create a contract for the underlyer (the S&P 500 index) and qualify them:

**Graphical user interface, text, application

Description automatically generated**

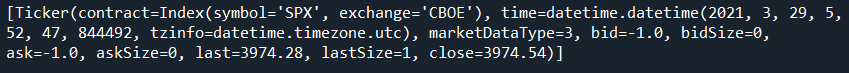
**Text

Description automatically generated**

* + - To avoid issues with market data permissions, we’ll use delayed data, this is specified by the integer 4, then get the ticker which can take up to 11 seconds

**Graphical user interface, text

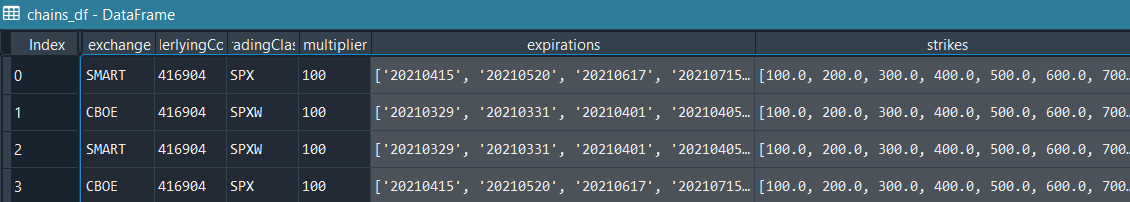
Description automatically generated**



* + - We can take the current market value of the ticker as well by using .marketPrice()



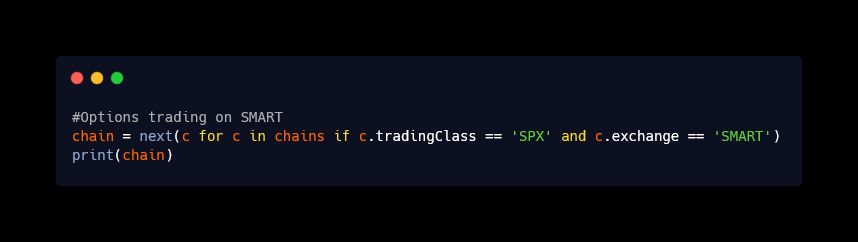
* + - We can then request a list of option chains and convert the to a Dataframe

Graphical user interface, text

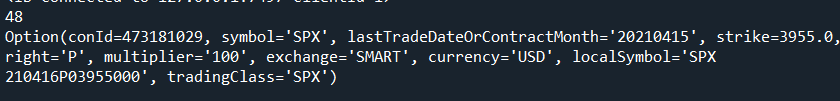
Description automatically generated

* + - There are four options that differ in exchange and tradingClass. SPX is for monthly and SPXW is for weekly options. Since we are interested only in the monthly options trading on SMART, we can create the following condition for it

Text

Description automatically generated

* + - We have a full matrix of expirations, from this we can build all the option contracts that meet our conditions: we want to use the next three-monthly expiries, use strike prices within +- $20 of the current SPX value, use strike prices that are a multitude of $5

Text

Description automatically generated

* + - We can then request market data for all the contracts, gives error, subscription required, delayed market data is available but don’t know how to access it

**Graphical user interface, text

Description automatically generated**

* **Market Depth (Order Book):**
  + To get a list of all exchanges that support market depth, we can use .reqMktDepthExchanges(), printing the first five shows:

**Text

Description automatically generated**

* + We can subscribe to a market depth, in this case we will subscribe to EURUSD

**Text

Description automatically generated**

* + To see a live order book, an event handler for tick updates is made that displays a dynamically updated Dataframe, we will run it for 1 second, we can then stop the market subscription by using cancelMkdtDepth(contract)

**Text

Description automatically generated**

Text

Description automatically generated

* **Scanners:**
  + **Basic Scanner:**
    - To create a scanner, create a ScannerSubscription to submit to the reqScannerData method. For any scanner to work at least three fields must be filled: instrument(what), locationCode(where) and scanCode(ranking)
    - For example to find the top ranked US stock percentage gainers, Historical Market Data Service error message, 0 results are obtained
* **Ordering:**
  + **Creating a Bracket Order:**
    - To create an order, we need to create a contract first, then an order and then we need to send a request to the IB API
    - Text

      Description automatically generatedThere are four type of orders, orders can be a MarketOrder, LimitOrder, StopOrder or StopLimtOrder
    - In the following image, we are creating three orders:
      * a LimitOrder, for one share in the Apple stock which says to buy the stock if the price is $123.10,
      * a LimitOrder, to sell the share to buy the stock if the price hit $125.50,
      * a StopOrder, to sell the share if the price drops to $122.50
    - We are using placeOrder to submit a request to the IB API. The following image show the TWS

A screenshot of a computer

Description automatically generated

* + - Similarly, we can create a Market Order as well and we check the status of the resulting trade.
    - We can also create a call back function which will update the trade status to filled once the order is executed
    - The following image shows that the order is Pending Submit since it was placed out of regular trading hours, and since the order has not been filled or executed, that property is shown as None

Text

Description automatically generated

* + **Creating Orders based on a Price Condition:**
    - We can create Market Order based on a Price Condition. In the following example, we intend to buy a stock for MasterCard when the stock for Visa hits a certain price
    - We start by creating a contract for Visa, next we create the Price Condition which has the following parameters:
      * price: this is the price Visa needs to hit to trigger our buy
      * conId: this is get automatically populated when we apply qualifyContracts on Visa
      * exch: the exchange on which the stock trades
    - The following image shows TWS after running the code, the clock means that there is a condition

