QuantConnect Documentation - Research Environment

Created on 05/17/2023

Table of Content

1 Key Concepts

1.1 Getting Started

Introduction

The Research Environment is a [Jupyter notebook](#) -based, interactive commandline environment where you can access our data through the `QuantBook` class. The environment supports both Python and C#. If you use Python, you can import code from the code files in your project into the Research Environment to aid development.

Before you run backtests, we recommend testing your hypothesis in the Research Environment. It's easier to perform data analysis and [produce plots in the Research Environment](#) than in a backtest.

Before backtesting or live trading with machine learning models, you may find it beneficial to train them in the Research Environment, save them in the ObjectStore, and then load them from the ObjectStore into the backtesting and live trading environment

In the Research Environment, you can also use the QuantConnect API to [import your backtest results](#) for further analysis.

Example

The following snippet demonstrates how to use the Research Environment to plot the price and Bollinger Bands of the S&P 500 index ETF, SPY:

The following snippet demonstrates how to use the Research Environment to print the price of the S&P 500 index ETF, SPY:

```
// Load the required assembly files and data types
#load "../Initialize.csx"
#load "../QuantConnect.csx"
using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;

// Create a QuantBook
var qb = new QuantBook();

// Create a security subscription
var symbol = qb.AddEquity("SPY").Symbol;

// Request some historical data
var history = qb.History(symbol, 70, Resolution.Daily);

foreach (var tradeBar in history)
{
    Console.WriteLine($"{tradeBar.EndTime} :: {tradeBar.ToString()}");
}
```

```
# Create a QuantBook
qb = QuantBook()

# Create a security subscription
spy = qb.AddEquity("SPY")

# Request some historical data
history = qb.History(qb.Securities.Keys, 360, Resolution.Daily)

# Calculate the Bollinger Bands
bbdf = qb.Indicator(BollingerBands(30, 2), spy.Symbol, 360, Resolution.Daily)

# Plot the data
bbdf.drop('standarddeviation', 1).plot()
```

Open Notebooks

Each new project you create contains a notebook file by default. Follow these steps to open the notebook:

1. [Open the project](#) .
2. In the right navigation menu, click the Explorer icon.
3. In the Explorer panel, expand the Workspace (Workspace) section.
4. Click the Research.ipynb research.ipynb file.

Run Notebook Cells

Notebooks are a collection of cells where you can execute code snippets or write MarkDown.

The following describes some helpful keyboard shortcuts to speed up your research:

Keyboard Shortcut	Description
Shift+Enter	Run the selected cell.
a	Insert a cell above the selected cell.
b	Insert a cell below the selected cell.
x	Cut the selected cell.
v	Paste the copied or cut cell.
z	Undo cell actions.

Stop Nodes

You need [stop node permissions](#) to stop research nodes in the cloud.

Follow these steps to stop a research node:

1. [Open the project](#) .
2. In the right navigation menu, click the Resources icon.
3. Click the stop button next to the research node you want to stop.

Add Notebooks

Follow these steps to add notebook files to a project:

1. [Open the project](#) .
2. In the right navigation menu, click the Explorer icon.
3. In the Explorer panel, expand the Workspace (Workspace) section.
4. Click the New File icon.
5. Enter fileName .ipynb .
6. Press Enter .

Rename Notebooks

Follow these steps to rename a notebook in a project:

1. Open the project .
2. In the right navigation menu, click the ▫ Explorer icon.
3. In the Explorer panel, right-click the notebook you want to rename and then click Rename .
4. Enter the new name and then press Enter .

Delete Notebooks

Follow these steps to delete a notebook in a project:

1. Open the project .
2. In the right navigation menu, click the ▫ Explorer icon.
3. In the Explorer panel, right-click the notebook you want to delete and then click Delete Permanently .
4. Click Delete .

Learn Jupyter

The following table lists some helpful resources to learn Jupyter:

Type	Name	Producer
Text	Jupyter Tutorial	tutorialspoint
Text	Jupyter Notebook Tutorial: The Definitive Guide	DataCamp

1.2 Research Engine

Introduction

The Research Environment is a [Jupyter notebook](#) -based, interactive commandline environment where you can access our data through the `QuantBook` class. The environment supports both Python and C#. If you use Python, you can import code from the code files in your project into the Research Environment to aid development.

Before you run backtests, we recommend testing your hypothesis in the Research Environment. It's easier to perform data analysis and [produce plots in the Research Environment](#) than in a backtest.

Before backtesting or live trading with machine learning models, you may find it beneficial to train them in the Research Environment, save them in the ObjectStore, and then load them from the ObjectStore into the backtesting and live trading environment

In the Research Environment, you can also use the QuantConnect API to [import your backtest results](#) for further analysis.

Batch vs Stream Analysis

The backtesting environment is an event-based simulation of the market. Backtests aim to provide an accurate representation of whether a strategy would have performed well in the past, but they are generally slow and aren't the most efficient way to test the foundational ideas behind strategies. You should only use backtests to verify an idea after you have already tested it with statistical analysis.

The Research Environment lets you build a strategy by starting with a central hypothesis about the market. For example, you might hypothesize that an increase in sunshine hours will increase the production of oranges, which will lead to an increase in the supply of oranges and a decrease in the price of Orange Juice Futures. You can attempt to confirm this working hypothesis by analyzing weather data, production of oranges data, and the price of Orange Juice futures. If the hypothesis is confirmed with a degree of statistical significance, you can be confident in the hypothesis and translate it into an algorithm you can backtest.

Jupyter Notebooks

Jupyter notebooks support interactive data science and scientific computing across various programming languages. We carry on that philosophy by providing an environment for you to perform exploratory research and brainstorm new ideas for algorithms. A Jupyter notebook installed in QuantConnect allows you to directly explore the massive amounts of data that is available in the Dataset Market and analyze it with python or C# commands. We call this exploratory notebook environment the Research Environment.

Open Notebooks

To open a notebook, open one of the .ipynb files in your [cloud projects](#) or see [Running Local Research Environment](#) .

Execute Code

The notebook allows you to run code in a safe and disposable environment. It's composed of independent cells where you can write, edit, and execute code. The notebooks support Python, C#, and Markdown code.

Keyboard Shortcuts

The following table describes some useful keyboard shortcuts:

Shortcut	Description
Shift+Enter	Run the selected cell
a	Insert a cell above the selected cell
b	Insert a cell below the selected cell
x	Cut the selected cell
v	Paste the copied or cut cell
z	Undo cell actions

Terminate Research Sessions

If you use the Research Environment in QuantConnect Cloud, to terminate a research session, stop the research node in the [Resources panel](#) . If you use the local Research Environment, see [Managing Kernels and Terminals](#) in the JupyterLab documentation.

Your Research and LEAN

To analyze data in a research notebook, create an instance of the `QuantBook` class. `QuantBook` is a wrapper on `QCAlgorithm` , which means `QuantBook` allows you to access all the methods available to `QCAlgorithm` and some additional methods. The following table describes the helper methods of the `QuantBook` class that aren't available in the `QCAlgorithm` class:

Method	Description
[GetFundamental](#)	Get fundamental data for some Symbol(s).
[GetFutureHistory](#)	Get the expiration, open interest, and price data of the contracts in a Futures chain.
[GetOptionHistory](#)	Get the strike, expiration, open interest, option right, and price data of the contracts in an Options chain.
[Indicator](#)	Get the values of an indicator for an asset over time.

QuantBook gives you access to the vast amounts of data in the Dataset Market. Similar to backtesting, you can access that data using history calls. You can also create indicators, consolidate data, and access charting features. However, keep in mind that event-driven features available in backtesting, like universe selection and OnData events, are not available in research. After you analyze a dataset in the Research Environment, you can easily transfer the logic to the backtesting environment. For example, consider the following code in the Research Environment:

```
// Initialize QuantBook
var qb = new QuantBook();

// Subscribe to SPY data with QuantBook
var symbol = qb.AddEquity("SPY").Symbol;

// Make history call with QuantBook
var history = qb.History(symbol, TimeSpan.FromDays(10), Resolution.Daily);
```

```
# Initialize QuantBook
qb = QuantBook()

# Subscribe to SPY data with QuantBook
symbol = qb.AddEquity("SPY").Symbol

# Make history call with QuantBook
history = qb.History(symbol, timedelta(days=10), Resolution.Daily)
```

To use the preceding code in a backtest, replace `QuantBook()` `new QuantBook()` with `self` `this` .

```
public override void Initialize()
{
    // Set qb to instance of QCAlgorithm
    var qb = this;

    // Subscribe to SPY data with QCAlgorithm
    var symbol = qb.AddEquity("SPY").Symbol;

    // Make history call with QCAlgorithm
    var history = qb.History(symbol, TimeSpan.FromDays(10), Resolution.Daily);
}
```

```
def Initialize(self) -> None:

    # Set qb to instance of QCAlgorithm
    qb = self

    # Subscribe to SPY data with QCAlgorithm
    symbol = qb.AddEquity("SPY").Symbol
```

```
    # Make history call with QCAlgorithm
    history = qb.History(symbol, timedelta(days=10), Resolution.Daily)
```

Import Project Code

To import code from your code files to your Research Environment session, use Python.

One of the drawbacks of using the Research Environment you may encounter is the need to rewrite code you've already written in a file in the backtesting environment. Instead of rewriting the code, you can import the methods from the backtesting environment into the Research Environment to reduce development time. For example, say you have the following helpers.py file in your project:

```
def Add(a, b):
   return a+b
```

To import the preceding method into your research notebook, use the `import` statement.

```
from helpers import Add

# reuse method from helpers.py
Add(3, 4)
```

If you adjust the file that you import, restart the Research Environment session to import the latest version of the file. To restart the Research Environment, stop the research node and then open the notebook again.

2 Initialization

Introduction

Before you request and manipulate historical data in the Research Environment, you should set the notebook dates, add data subscriptions, and set the time zone.

Set Dates

The start date of your `QuantBook` determines the latest date of data you get from [history requests](). By default, the start date is the current day. To change the start date, call the `SetStartDate` method.

```
qb.SetStartDate(2022, 1, 1);
```

```
qb.SetStartDate(2022, 1, 1)
```

The end date of your `QuantBook` should be greater than the end date. By default, the start date is the current day. To change the end date, call the `SetEndDate` method.

```
qb.SetEndDate(2022, 8, 15);
```

```
qb.SetEndDate(2022, 8, 15)
```

Add Data

You can subscribe to asset, fundamental, alternative, and custom data. The [Dataset Market]() provides 400TB of data that you can easily import into your notebooks.

Asset Data

To subscribe to asset data, call one of the asset subscription methods like `AddEquity` or `AddForex` . Each asset class has its own method to create subscriptions. For more information about how to create subscriptions for each asset class, see the Create Subscriptions section of an asset class in the [Datasets]() chapter.

```
qb.AddEquity("AAPL"); // Add Apple 1 minute bars (minute by default)
qb.AddForex("EURUSD", Resolution.Second); // Add EURUSD 1 second bars
```

```
qb.AddEquity("SPY")  # Add Apple 1 minute bars (minute by default)
qb.AddForex("EURUSD", Resolution.Second) # Add EURUSD 1 second bars
```

Alternative Data

To add alternative datasets to your notebooks, call the `AddData` method. For a full example, see [Alternative Data]().

Custom Data

To add custom data to your notebooks, call the `AddData` method. For more information about custom data, see [Custom Data]().

Limitations

There is no official limit to how much data you can add to your notebooks, but there are practical resource limitations. Each security subscription requires about 5MB of RAM, so larger machines let you request more data. For more information about our cloud nodes, see [Research Nodes]().

Set Time Zone

The notebook time zone determines which time zone the `DateTime` `datetime` objects are in when you make a [history request]() based on a defined period of time. When your history request returns a `DataFrame` , the timestamps in the `DataFrame` are based on the [data time zone](). When your history request returns a `TradeBars` , `QuoteBars` , `Ticks` , or `Slice` object, the `Time` properties of these objects are based on the notebook time zone, but the `EndTime` properties of the individual `TradeBar` , `QuoteBar` , and `Tick` objects are based on the data time zone.

The default time zone is Eastern Time (ET), which is UTC-4 in summer and UTC-5 in winter. To set a different time zone, call the `SetTimeZone` method. This method accepts either a string following the [IANA Time Zone database]() convention or a [NodaTime]().DateTimeZone object. If you pass a string, the method converts it to a `NodaTime.DateTimeZone` object. The `TimeZones` class provides the following helper attributes to create `NodaTime.DateTimeZone` objects:

```
SetTimeZone("Europe/London");
qb.SetTimeZone(NodaTime.DateTimeZone.Utc);
qb.SetTimeZone(TimeZones.Chicago);
```

```
qb.SetTimeZone("Europe/London")
qb.SetTimeZone(TimeZones.Chicago)
```

3 Datasets

3.1 Key Concepts

Introduction

You can access most of the data from the Dataset Market in the Research Environment. The data includes Equity, Crypto, Forex, and derivative data going back as far as 1998. Similar to backtesting, to access the data, create a security subscription and then make a history request.

Key History Concepts

The historical data API has many different options to give you the greatest flexibility in how to apply it to your algorithm.

Time Period Options

You can request historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. If you request data in a defined period of time, the `DateTime` `datetime` objects you provide are based in the notebook time zone .

Return Formats

Each asset class supports slightly different data formats. When you make a history request, consider what data returns. Depending on how you request the data, history requests return a specific data type. For example, if you don't provide `Symbol` objects, you get `Slice` objects that contain all of the assets you created subscriptions for in the notebook.

The most popular return type is a `DataFrame` . If you request a `DataFrame` , LEAN unpacks the data from `Slice` objects to populate the `DataFrame` . If you intend to use the data in the `DataFrame` to create `TradeBar` or `QuoteBar` objects, request that the history request returns the data type you need. Otherwise, LEAN will waste computational resources populating the `DataFrame` .

Time Index

When your history request returns a `DataFrame` , the timestamps in the `DataFrame` are based on the data time zone . When your history request returns a `TradeBars` , `QuoteBars` , `Ticks` , or `Slice` object, the `Time` properties of these objects are based on the notebook time zone, but the `EndTime` properties of the individual `TradeBar` , `QuoteBar` , and `Tick` objects are based on the data time zone data time zone . The `EndTime` is the end of the sampling period and when the data is actually available. For daily US Equity data, this results in data points appearing on Saturday and skipping Monday.

Request Data

The simplest form of history request is for a known set of `Symbol` objects. History requests return slightly different data depending on the overload you call. The data that returns is in ascending order from oldest to newest.

Single Symbol History Requests

To request history for a single asset, pass the asset `Symbol` to the `History` method. The return type of the method call depends on the history request `[Type]` `<Type>` . The following table describes the return type of each request `[Type]` `<Type>` :

Request Type	Return Data Type
No argument	`DataFrame` `List<TradeBar>`
`TradeBar`	`List[TradeBars]` `List<TradeBar>`
`QuoteBar`	`List[QuoteBars]` `List<QuoteBar>`
`Tick`	`List[Ticks]` `List<Tick>`
`alternativeDataClass` (ex: `CBOE`)	`List[ alternativeDataClass ]` (ex: `List[CBOE]`) `List< alternativeDataClass >` (ex: `List<CBOE>`)

Each row of the DataFrame represents the prices at a point in time. Each column of the DataFrame is a property of that price data (for example, open, high, low, and close (OHLC)). If you request a DataFrame object and pass `TradeBar` as the first argument, the DataFrame that returns only contains the OHLC and volume columns. If you request a DataFrame object and pass `QuoteBar` as the first argument, the DataFrame that returns contains the OHLC of the bid and ask and it contains OHLC columns, which are the respective means of the bid and ask OHLC values. If you request a DataFrame and don't pass `TradeBar` or `QuoteBar` as the first arugment, the DataFrame that returns contains columns for all of the data that's available for the given resolution.

```
# EXAMPLE 1: Requesting By Bar Count: 5 bars at the security resolution:
vix_symbol = qb.AddData(CBOE, "VIX", Resolution.Daily).Symbol
cboe_data = qb.History[CBOE](vix_symbol, 5)

btc_symbol = qb.AddCrypto("BTCUSD", Resolution.Minute).Symbol
trade_bars = qb.History[TradeBar](btc_symbol, 5)
quote_bars = qb.History[QuoteBar](btc_symbol, 5)
trade_bars_df = qb.History(TradeBar, btc_symbol, 5)
quote_bars_df = qb.History(QuoteBar, btc_symbol, 5)
df = qb.History(btc_symbol, 5)   # Includes trade and quote data
```

```
// EXAMPLE 1: Requesting By Bar Count: 5 bars at the security resolution:
var vixSymbol = qb.AddData<CBOE>("VIX", Resolution.Daily).Symbol;
var cboeData = qb.History<CBOE>(vixSymbol, 5);

var btcSymbol = qb.AddCrypto("BTCUSD", Resolution.Minute).Symbol;
var tradeBars = qb.History<TradeBar>(btcSymbol, 5);
var quoteBars = qb.History<QuoteBar>(btcSymbol, 5);
var tradeBars2 = qb.History(btcSymbol, 5);
```

```
# EXAMPLE 2: Requesting By Bar Count: 5 bars with a specific resolution:
trade_bars = qb.History[TradeBar](btc_symbol, 5, Resolution.Daily)
quote_bars = qb.History[QuoteBar](btc_symbol, 5, Resolution.Minute)
trade_bars_df = qb.History(TradeBar, btc_symbol, 5, Resolution.Minute)
quote_bars_df = qb.History(QuoteBar, btc_symbol, 5, Resolution.Minute)
df = qb.History(btc_symbol, 5, Resolution.Minute)  # Includes trade and quote data
```

```
// EXAMPLE 2: Requesting By Bar Count: 5 bars with a specific resolution:
var tradeBars = qb.History<TradeBar>(btcSymbol, 5, Resolution.Daily);
var quoteBars = qb.History<QuoteBar>(btcSymbol, 5, Resolution.Minute);
var tradeBars2 = qb.History(btcSymbol, 5, Resolution.Minute);
```

```
# EXAMPLE 3: Requesting By a Trailing Period: 3 days of data at the security resolution:
eth_symbol = qb.AddCrypto('ETHUSD', Resolution.Tick).Symbol
ticks = qb.History[Tick](eth_symbol, timedelta(days=3))
ticks_df = qb.History(eth_symbol, timedelta(days=3))

vix_data = qb.History[CBOE](vix_symbol, timedelta(days=3))
trade_bars = qb.History[TradeBar](btc_symbol, timedelta(days=3))
quote_bars = qb.History[QuoteBar](btc_symbol, timedelta(days=3))
trade_bars_df = qb.History(TradeBar, btc_symbol, timedelta(days=3))
quote_bars_df = qb.History(QuoteBar, btc_symbol, timedelta(days=3))
df = qb.History(btc_symbol, timedelta(days=3))  # Includes trade and quote data
```

```
// EXAMPLE 3: Requesting By a Trailing Period: 3 days of data at the security resolution:
var ethSymbol = qb.AddCrypto("ETHUSD", Resolution.Tick).Symbol;
var ticks = qb.History<Tick>(ethSymbol, TimeSpan.FromDays(3));

var cboeData = qb.History<CBOE>(vixSymbol, TimeSpan.FromDays(3));
var tradeBars = qb.History<TradeBar>(btcSymbol, TimeSpan.FromDays(3));
var quoteBars = qb.History<QuoteBar>(btcSymbol, TimeSpan.FromDays(3));
var tradeBars2 = qb.History(btcSymbol, TimeSpan.FromDays(3));
```

```
# EXAMPLE 4: Requesting By a Trailing Period: 3 days of data with a specific resolution:
trade_bars = qb.History[TradeBar](btc_symbol, timedelta(days=3), Resolution.Daily)
quote_bars = qb.History[QuoteBar](btc_symbol, timedelta(days=3), Resolution.Minute)
ticks = qb.History[Tick](eth_symbol, timedelta(days=3), Resolution.Tick)

trade_bars_df = qb.History(TradeBar, btc_symbol, timedelta(days=3), Resolution.Daily)
```

```
quote_bars_df = qb.History(QuoteBar, btc_symbol, timedelta(days=3), Resolution.Minute)
ticks_df = qb.History(eth_symbol, timedelta(days=3), Resolution.Tick)
df = qb.History(btc_symbol, timedelta(days=3), Resolution.Hour)  # Includes trade and quote data
```

```
# Important Note: Period history requests are relative to "now" notebook time.
```

```
// EXAMPLE 4: Requesting By a Trailing Period: 3 days of data with a specific resolution:
var tradeBars = qb.History<TradeBar>(btcSymbol, TimeSpan.FromDays(3), Resolution.Daily);
var quoteBars = qb.History<QuoteBar>(btcSymbol, TimeSpan.FromDays(3), Resolution.Minute);
var ticks = qb.History<Tick>(ethSymbol, TimeSpan.FromDays(3), Resolution.Tick);
var tradeBars2 = qb.History(btcSymbol, TimeSpan.FromDays(3), Resolution.Minute);
```

```
# EXAMPLE 5: Requesting By a Defined Period: 3 days of data at the security resolution:
start_time = datetime(2022, 1, 1)
end_time = datetime(2022, 1, 4)

vix_data = qb.History[CBOE](vix_symbol, start_time, end_time)
trade_bars = qb.History[TradeBar](btc_symbol, start_time, end_time)
quote_bars = qb.History[QuoteBar](btc_symbol, start_time, end_time)
ticks = qb.History[Tick](eth_symbol, start_time, end_time)

trade_bars_df = qb.History(TradeBar, btc_symbol, start_time, end_time)
quote_bars_df = qb.History(QuoteBar, btc_symbol, start_time, end_time)
ticks_df = qb.History(Tick, eth_symbol, start_time, end_time)
df = qb.History(btc_symbol, start_time, end_time)  # Includes trade and quote data
```

```
// EXAMPLE 5: Requesting By a Defined Period: 3 specific days of data at the security resolution:
var startTime = new DateTime(2022, 1, 1);
var endTime = new DateTime(2022, 1, 4);

var cboeData = qb.History<CBOE>(vixSymbol, startTime, endTime);
var tradeBars = qb.History<TradeBar>(btcSymbol, startTime, endTime);
var quoteBars = qb.History<QuoteBar>(btcSymbol, startTime, endTime);
var ticks = qb.History<Tick>(ethSymbol, startTime, endTime);
var tradeBars2 = qb.History(btcSymbol, startTime, endTime);
```

```
# EXAMPLE 6: Requesting By a Defined Period: 3 days of data with a specific resolution:
trade_bars = qb.History[TradeBar](btc_symbol, start_time, end_time, Resolution.Daily)
quote_bars = qb.History[QuoteBar](btc_symbol, start_time, end_time, Resolution.Minute)
ticks = qb.History[Tick](eth_symbol, start_time, end_time, Resolution.Tick)

trade_bars_df = qb.History(TradeBar, btc_symbol, start_time, end_time, Resolution.Daily)
quote_bars_df = qb.History(QuoteBar, btc_symbol, start_time, end_time, Resolution.Minute)
ticks_df = qb.History(eth_symbol, start_time, end_time, Resolution.Tick)
df = qb.History(btc_symbol, start_time, end_time, Resolution.Hour)  # Includes trade and quote data
```

```
// EXAMPLE 6: Requesting By a Defined Period: 3 days of data with a specific resolution:
var tradeBars = qb.History<TradeBar>(btcSymbol, startTime, endTime, Resolution.Daily);
var quoteBars = qb.History<QuoteBar>(btcSymbol, startTime, endTime, Resolution.Minute);
var ticks = qb.History<Tick>(ethSymbol, startTime, endTime, Resolution.Tick);
var tradeBars2 = qb.History(btcSymbol, startTime, endTime, Resolution.Minute);
```

Multiple Symbol History Requests

To request history for multiple symbols at a time, pass an array of `Symbol` objects to the same API methods shown in the preceding section. The return type of the method call depends on the history request `[Type]` `<Type>` . The following table describes the return type of each request `[Type]` `<Type>` :

Request Type	Return Data Type
No argument	`DataFrame` `List<Slice>`
`TradeBar`	`List[TradeBars]` `List<TradeBars>`
`QuoteBar`	`List[QuoteBars]` `List<QuoteBars>`
`Tick`	`List[Ticks]` `List<Ticks>`
`alternativeDataClass` (ex: `CBOE`)	`List[Dict[Symbol, alternativeDataClass ]]` (ex: `List[Dict[Symbol, CBOE]]`) `List<Dictionary<Symbol, alternativeDataClass >>` (ex: `List<Dictionary<Symbol, CBOE>>`)

The `Slice` return type provides a container that supports all data types. For example, a history request for Forex `QuoteBars` and Equity `TradeBars` has the Forex data under `slices.QuoteBars` and the Equity data under `slices.Bars` .

```
# EXAMPLE 7: Requesting By Bar Count for Multiple Symbols: 2 bars at the security resolution:
vix = qb.AddData[CBOE]("VIX", Resolution.Daily).Symbol
v3m = qb.AddData[CBOE]("VIX3M", Resolution.Daily).Symbol
cboe_data = qb.History[CBOE]([vix, v3m], 2)

ibm = qb.AddEquity("IBM", Resolution.Minute).Symbol
aapl = qb.AddEquity("AAPL", Resolution.Minute).Symbol
trade_bars_list = qb.History[TradeBar]([ibm, aapl], 2)
quote_bars_list = qb.History[QuoteBar]([ibm, aapl], 2)

trade_bars_df = qb.History(TradeBar, [ibm, aapl], 2)
quote_bars_df = qb.History(QuoteBar, [ibm, aapl], 2)
df = qb.History([ibm, aapl], 2)  # Includes trade and quote data
```

```
// EXAMPLE 7: Requesting By Bar Count for Multiple Symbols: 2 bars at the security resolution:
var vixSymbol = qb.AddData<CBOE>("VIX", Resolution.Daily).Symbol;
var v3mSymbol = qb.AddData<CBOE>("VIX3m", Resolution.Daily).Symbol;
var cboeData = qb.History<CBOE>(new[] { vix, v3m }, 2);

var ibm = qb.AddEquity("IBM", Resolution.Minute).Symbol;
var aapl = qb.AddEquity("AAPL", Resolution.Minute).Symbol;
var tradeBarsList = qb.History<TradeBar>(new[] { ibm, aapl }, 2);
var quoteBarsList = qb.History<QuoteBar>(new[] { ibm, aapl }, 2);
```

```
# EXAMPLE 8: Requesting By Bar Count for Multiple Symbols: 5 bars with a specific resolution:
trade_bars_list = qb.History[TradeBar]([ibm, aapl], 5, Resolution.Daily)
quote_bars_list = qb.History[QuoteBar]([ibm, aapl], 5, Resolution.Minute)

trade_bars_df = qb.History(TradeBar, [ibm, aapl], 5, Resolution.Daily)
quote_bars_df = qb.History(QuoteBar, [ibm, aapl], 5, Resolution.Minute)
df = qb.History([ibm, aapl], 5, Resolution.Daily)  # Includes trade data only. No quote for daily equity data
```

```
// EXAMPLE 8: Requesting By Bar Count for Multiple Symbols: 5 bars with a specific resolution:
var tradeBarsList = qb.History<TradeBar>(new[] { ibm, aapl }, 5, Resolution.Minute);
var quoteBarsList = qb.History<QuoteBar>(new[] { ibm, aapl }, 5, Resolution.Minute);
```

```
# EXAMPLE 9: Requesting By Trailing Period: 3 days of data at the security resolution:
ticks = qb.History[Tick]([eth_symbol], timedelta(days=3))

trade_bars = qb.History[TradeBar]([btc_symbol], timedelta(days=3))
quote_bars = qb.History[QuoteBar]([btc_symbol], timedelta(days=3))
trade_bars_df = qb.History(TradeBar, [btc_symbol], timedelta(days=3))
quote_bars_df = qb.History(QuoteBar, [btc_symbol], timedelta(days=3))
df = qb.History([btc_symbol], timedelta(days=3))  # Includes trade and quote data
```

```
// EXAMPLE 9: Requesting By Trailing Period: 3 days of data at the security resolution:
var ticks = qb.History<Tick>(new[] {ethSymbol}, TimeSpan.FromDays(3));

var tradeBars = qb.History<TradeBar>(new[] {btcSymbol}, TimeSpan.FromDays(3));
var quoteBars = qb.History<QuoteBar>(new[] {btcSymbol}, TimeSpan.FromDays(3));
var tradeBars2 = qb.History(new[] {btcSymbol}, TimeSpan.FromDays(3));
```

```
# EXAMPLE 10: Requesting By Defined Period: 3 days of data at the security resolution:
trade_bars = qb.History[TradeBar]([btc_symbol], start_time, end_time)
quote_bars = qb.History[QuoteBar]([btc_symbol], start_time, end_time)
ticks = qb.History[Tick]([eth_symbol], start_time, end_time)
trade_bars_df = qb.History(TradeBar, btc_symbol, start_time, end_time)
quote_bars_df = qb.History(QuoteBar, btc_symbol, start_time, end_time)
ticks_df = qb.History(Tick, eth_symbol, start_time, end_time)
df = qb.History([btc_symbol], start_time, end_time)  # Includes trade and quote data
```

```
// EXAMPLE 10: Requesting By Defined Period: 3 days of data at the security resolution:
var tradeBars = qb.History<TradeBar>(new[] {btcSymbol}, startTime, endTime);
var quoteBars = qb.History<QuoteBar>(new[] {btcSymbol}, startTime, endTime);
var ticks = qb.History<Tick>(new[] {ethSymbol}, startTime, endTime);
var tradeBars2 = qb.History(new[] {btcSymbol}, startTime, endTime);
```

All Symbol History Requests

You can request history for all the securities you have created subscriptions for in your notebook session. The parameters are very similar to other history method calls, but the return type is an array of Slice objects. The `Slice` object holds all of the results in a sorted enumerable collection that you can iterate over with a loop.

```
# EXAMPLE 11: Requesting 5 bars for all securities at their respective resolution:

# Create subscriptions
qb.AddEquity("IBM", Resolution.Daily)
qb.AddEquity("AAPL", Resolution.Daily)

# Request history data and enumerate results
slices = qb.History(5)
for s in slices:
    print(str(s.Time) + " AAPL:" + str(s.Bars["AAPL"].Close) + " IBM:" + str(s.Bars["IBM"].Close))
```

```
// EXAMPLE 11: Requesting 5 bars for all securities at their respective resolution:

// Set up the universe
qb.AddEquity("IBM", Resolution.Daily);
qb.AddEquity("AAPL", Resolution.Daily);

// Request history data and enumerate results:
var slices = qb.History(5);
foreach (var s in slices) {
    var aaplClose = s.Bars["AAPL"].Close;
    var ibmClose = s.Bars["IBM"].Close;
    Console.WriteLine($"{s.Time} AAPL: {aaplClose} IBM: {ibmClose}");
}
```

```
# EXAMPLE 12: Requesting 5 minutes for all securities:

slices = qb.History(timedelta(minutes=5), Resolution.Minute)
for s in slices:
    print(str(s.Time) + " AAPL:" + str(s.Bars["AAPL"].Close) + " IBM:" + str(s.Bars["IBM"].Close))

# timedelta history requests are relative to "now" in notebook Time. If you request this data at 16:05, it returns an empty array because the market is closed.
```

```
// EXAMPLE 12: Requesting 24 hours of hourly data for all securities:

var slices = qb.History(TimeSpan.FromHours(24), Resolution.Hour);
foreach (var s in slices) {
    var aaplClose = s.Bars["AAPL"].Close;
    var ibmClose = s.Bars["IBM"].Close;
    Console.WriteLine($"{s.Time} AAPL: {aaplClose} IBM: {ibmClose}");
}

// TimeSpan history requests are relative to "now" in notebook Time.
```

Assumed Default Values

The following table describes the assumptions of the History API:

Argument	Assumption
Resolution	LEAN guesses the resolution you request by looking at the securities you already have in your notebook. If you have a security subscription in your notebook with a matching `Symbol`, the history request uses the same resolution as the subscription. If you don't have a security subscription in your notebook with a matching `Symbol`, `Resolution.Minute` is the default.
Bar type	If you don't specify a type for the history request, `TradeBar` is the default. If the asset you request data for doesn't have `TradeBar` data, specify the `QuoteBar` type to receive history.

Additional Options

The `History` method accepts the following additional arguments:

Argument	Data Type	Description	Default Value
`fillForward`	`bool? bool/NoneType`	True to fill forward missing data. Otherwise, false.	`null None`
`extendedMarketHours`	`bool? bool/NoneType`	True to include extended market hours data. Otherwise, false.	`null None`
`dataMappingMode`	`DataMappingMode? DataMappingMode/NoneType`	The contract mapping mode to use for the security history request.	`null None`
`dataNormalizationMode`	`DataNormalizationMode? DataNormalizationMode/NoneType`	The price scaling mode to use for US Equities or continuous Futures contracts. If you don't provide a value, it uses the data normalization mode of the security subscription.	`null None`
`contractDepthOffset`	`int? int/NoneType`	The desired offset from the current front month for continuous Futures contracts.	`null None`

```
future = qb.AddFuture(Futures.Currencies.BTC)
history = qb.History(
    tickers=[future.Symbol],
    start=qb.Time - timedelta(days=15),
    end=qb.Time,
    resolution=Resolution.Minute,
    fillForward=False,
    extendedMarketHours=False,
    dataMappingMode=DataMappingMode.OpenInterest,
    dataNormalizationMode=DataNormalizationMode.Raw,
    contractDepthOffset=0)
```

```
var future = qb.AddFuture(Futures.Currencies.BTC);
var history = qb.History(
    symbols: new[] {future.Symbol},
    start: qb.Time - TimeSpan.FromDays(15),
    end: qb.Time,
    resolution: Resolution.Minute,
    fillForward: false,
    extendedMarketHours: false,
    dataMappingMode: DataMappingMode.OpenInterest,
    dataNormalizationMode: DataNormalizationMode.Raw,
    contractDepthOffset: 0);
```

Resolutions

Resolution is the duration of time that's used to sample a data source. The `Resolution` enumeration has the following members:

The default resolution for market data is `Minute`. To set the resolution for a security, pass the `resolution` argument when you create the security subscription.

```
qb.AddEquity("SPY", Resolution.Daily);
```

```
qb.AddEquity("SPY", Resolution.Daily)
```

When you request historical data, the `History` method uses the resolution of your security subscription. To get historical data with a different resolution, pass a `resolution` argument to the `History` method.

```
history = qb.History(spy, 10, Resolution.Minute)
```

```
var history = qb.History(spy, 10, Resolution.Minute);
```

Markets

The datasets integrated into the Dataset Market cover many markets. The `Market` enumeration has the following members:

LEAN can usually determine the correct market based on the ticker you provide when you create the security subscription. To manually set the market for a security, pass a `market` argument when you create the security subscription.

```
qb.AddEquity("SPY", market: Market.USA);
```

```
qb.AddEquity("SPY", market=Market.USA)
```

Fill Forward

Fill forward means if there is no data point for the current sample, LEAN uses the previous data point. Fill forward is the default data setting. To disable fill forward for a security, set the `fillForward` argument to false when you create the security subscription.

```
qb.AddEquity("SPY", fillForward: false);
```

```
qb.AddEquity("SPY", fillForward=False)
```

When you request historical data, the `History` method uses the fill forward setting of your security subscription. To get historical data with a different fill forward setting, pass a `fillForward` argument to the `History` method.

```
var history = qb.History(qb.Securities.Keys, qb.Time-TimeSpan.FromDays(10), qb.Time, fillForward: true);
```

```
history = qb.History(qb.Securities.Keys, qb.Time-timedelta(days=10), qb.Time, fillForward=True)
```

Extended Market Hours

By default, your security subscriptions only cover regular trading hours. To subscribe to pre and post-market trading hours for a specific asset, enable the `extendedMarketHours` argument when you create the security subscription.

```
AddEquity("SPY", extendedMarketHours: true);
```

```
self.AddEquity("SPY", extendedMarketHours=True)
```

You only receive extended market hours data if you create the subscription with minute, second, or tick resolution. If you create the subscription with daily or hourly resolution, the bars only reflect the regular trading hours.

When you request historical data, the `History` method uses the extended market hours setting of your security subscription. To get historical data with a different extended market hours setting, pass an `extendedMarketHours` argument to the `History` method.

```
var history = qb.History(qb.Securities.Keys, qb.Time-TimeSpan.FromDays(10), qb.Time, extendedMarketHours: false);
```

```
history = qb.History(qb.Securities.Keys, qb.Time-timedelta(days=10), qb.Time, extendedMarketHours=False)
```

Look-Ahead Bias

In the Research Environment, all the historical data is directly available. In backtesting, you can only access the data that is at or before the algorithm time. If you make a history request for the previous 10 days of data in the Research Environment, you get the previous 10 days of data from today's date. If you request the same data in a backtest, you get the previous 10 days of data from the algorithm time.

Consolidate Data

History requests usually return data in one of the standard resolutions . To analyze data on custom time frames like 5-minute bars or 4-hour bars, you need to aggregate it. Consider an example where you make a history call for minute resolution data and want to create 5-minute resolution data.

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
start_date = datetime(2018, 4, 1)
end_date = datetime(2018, 7, 15)
history = qb.History(symbol, start_date, end_date, Resolution.Minute)
```

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
var startDate = new DateTime(2018, 4, 1);
var endDate = new DateTime(2018, 7, 15);
var history = qb.History(symbol, startDate, endDate, Resolution.Minute);
foreach (var slice in history)
{
    foreach (var key in slice.Keys)
    {
        Console.WriteLine($"{slice.Time} :: {slice[key].ToString()}");
    }
}
```

To aggregate the data, use a consolidator or the `pandas` resample method.

To aggregate the data, use a consolidator .

Consolidators

The following snippet demonstrates how to use a consolidator to aggregate data:

```
// Set up a consolidator and a RollingWindow to save the data
var consolidator = new TradeBarConsolidator(TimeSpan.FromDays(7));
var window = new RollingWindow<TradeBar>(20);

// Attach a consolidation handler method that saves the consolidated bars in the RollingWindow
consolidator.DataConsolidated += (sender, consolidated) =>
{
    window.Add(consolidated);
};

// Iterate the historical market data and feed each bar into the consolidator
foreach(var bar in history)
{
    consolidator.Update(bar);
}
```

```
# Set up a consolidator and a RollingWindow to save the data
consolidator = TradeBarConsolidator(timedelta(7))
window = RollingWindow[TradeBar](20)

# Attach a consolidation handler method that saves the consolidated bars in the RollingWindow
consolidator.DataConsolidated += lambda _, bar: window.Add(bar)

# Iterate the historical market data and feed each bar into the consolidator
for bar in history.itertuples():
    tradebar = TradeBar(bar.Index[1], bar.Index[0], bar.open, bar.high, bar.low, bar.close, bar.volume)
    consolidator.Update(tradebar)
```

Resample Method

The `resample` method converts the frequency of a time series DataFrame into a custom frequency. The method only works on DataFrame objects that have a `datetime` index. The `History` method returns a DataFrame with a multi-index. The first index is a `Symbol` index for each security and the second index is a time index for the timestamps of each row of data. To make the DataFrame compatible with the `resample` method, call the reset_index method to drop the `Symbol` index.

```
# Drop level 0 index (Symbol index) from the DataFrame
history.reset_index(level = 0, drop = True, inplace=True)
```

The `resample` method returns a `Resampler` object, which needs to be downsampled using one of the pandas downsampling computations . For example, you can use the Resampler.ohlc downsampling method to aggregate price data.

When you resample a DataFrame with the `ohlc` downsampling method, it creates an OHLC row for each column in the DataFrame. To just calculate the OHLC of the close column, select the close column before you resample the DataFrame. A resample offset of 5T corresponds to a 5-minute resample. Other resampling offsets include 2D = 2 days, 5H = 5 hours, and 3S = 3 seconds.

```
close_prices = history["close"]

offset = "5T"
close_5min_ohlc = close_prices.resample(offset).ohlc()
```

Common Errors

If the history request returns an empty DataFrame and you try to slice it, it throws an exception. To avoid issues, check if the DataFrame contains data before slicing it.

```
df = qb.History(symbol, 10).close    # raises exception if the request is empty

def GetSafeHistoryCloses(symbols):
    if not symbols:
        print(f'No symbols')
        return  False, None
    df = qb.History(symbols, 100, Resolution.Daily)
    if df.empty:
        print(f'Empy history for {symbols}')
        return  False, None
     return True, df.close.unstack(0)
```

If you run the Research Environment on your local machine and history requests return no data, check if your data directory contains the data you request. To download datasets, see Download .

3.2 US Equity

Introduction

This page explains how to request, manipulate, and visualize historical US Equity data.

Create Subscriptions

Follow these steps to subscribe to a US Equity security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
using QuantConnect.Indicators;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddEquity` method with a ticker and then save a reference to the US Equity `Symbol` .

```
var spy = qb.AddEquity("SPY").Symbol;
var tlt = qb.AddEquity("TLT").Symbol;
```

```
spy = qb.AddEquity("SPY").Symbol
tlt = qb.AddEquity("TLT").Symbol
```

To view the supported assets in the US Equities dataset, see the Data Explorer .

Get Historical Data

You need a subscription before you can request historical data for a security. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the security dimension, you can request historical data for a single US Equity, a subset of the US Equities you created subscriptions for in your notebook, or all of the US Equities in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(spy, 10);
var subsetHistorySlice = qb.History(new[] {spy, tlt}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spy, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spy, tlt}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spy, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spy, tlt}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(spy, 10)
subset_history_df = qb.History([spy, tlt], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, spy, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [spy, tlt], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, spy, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [spy, tlt], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spy, 10)
subset_history_trade_bars = qb.History[TradeBar]([spy, tlt], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spy, 10)
subset_history_quote_bars = qb.History[QuoteBar]([spy, tlt], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(spy, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {spy, tlt}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spy, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spy, tlt}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spy, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spy, tlt}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spy, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spy, tlt}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(spy, timedelta(days=3))
subset_history_df = qb.History([spy, tlt], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, spy, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [spy, tlt], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))
```

```
# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, spy, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [spy, tlt], timedelta(days=3))
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(spy, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([spy, tlt], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spy, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([spy, tlt], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spy, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([spy, tlt], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](spy, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spy, tlt], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

The preceding calls return the most recent bars or ticks, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime datetime` , and an end `DateTime datetime` . The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 2, 1);

// Slice objects
var singleHistorySlice = qb.History(spy, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {spy, tlt}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spy, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spy, tlt}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spy, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spy, tlt}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spy, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spy, tlt}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of trade and quote data
single_history_df = qb.History(spy, start_time, end_time)
subset_history_df = qb.History([spy, tlt], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, spy, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [spy, tlt], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, spy, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [spy, tlt], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(spy, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([spy, tlt], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spy, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([spy, tlt], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spy, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([spy, tlt], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](spy, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spy, tlt], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

The preceding calls return the bars or ticks that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Equity subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`			☐	☐
`Second`	☐	☐		
`Minute`	☐	☐		
`Hour`	☐			
`Daily`	☐			

Markets

LEAN groups all of the US Equity exchanges under `Market.USA` .

Data Normalization

The data normalization mode defines how historical data is adjusted for [corporate actions](#) . By default, LEAN adjusts US Equity data for splits and dividends to produce a smooth price curve, but the following data normalization modes are available:

We use the entire split and dividend history to adjust historical prices. This process ensures you get the same adjusted prices, regardless of the `QuantBook` time.

To set the data normalization mode for a security, pass a `dataNormalizationMode` argument to the `AddEquity` method.

```
var spy = qb.AddEquity("SPY", dataNormalizationMode: DataNormalizationMode.Raw).Symbol;
```

```
spy = qb.AddEquity("SPY", dataNormalizationMode=DataNormalizationMode.Raw).Symbol
```

When you request historical data, the `History` method uses the data normalization of your security subscription. To get historical data with a different data normalization, pass a `dataNormalizationMode` argument to the `History` method.

```
var history = qb.History(qb.Securities.Keys, qb.Time-TimeSpan.FromDays(10), qb.Time, dataNormalizationMode: DataNormalizationMode.SplitAdjusted);
```

```
history = qb.History(qb.Securities.Keys, qb.Time-timedelta(days=10), qb.Time, dataNormalizationMode=DataNormalizationMode.SplitAdjusted)
```

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame`, the first level of the `DataFrame` index is the encoded Equity Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single Equity, index the `loc` property of the `DataFrame` with the Equity `Symbol`.

```
all_history_df.loc[spy]  # or all_history_df.loc['SPY']
```

To select a column of the `DataFrame`, index it with the column name.

```
all_history_df.loc[spy]['close']
```

If you request historical data for multiple Equities, you can transform the `DataFrame` so that it's a time series of close values for all of the Equities. To transform the `DataFrame`, select the column you want to display for each Equity and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each Equity and each row contains the close value.

If you prefer to display the ticker of each `Symbol` instead of the string representation of the `SecurityIdentifier`, follow these steps:

1. Create a dictionary where the keys are the string representations of each `SecurityIdentifier` and the values are the ticker.

   ```
   tickers_by_id = {str(x.ID): x.Value for x in qb.Securities.Keys}
   ```

2. Get the values of the symbol level of the `DataFrame` index and create a list of tickers.

   ```
   tickers = set([tickers_by_id[x] for x in all_history_df.index.get_level_values('symbol')])
   ```

3. Set the values of the symbol level of the `DataFrame` index to the list of tickers.

   ```
   all_history_df.index.set_levels(tickers, 'symbol', inplace=True)
   ```

The new `DataFrame` is keyed by the ticker.

```
all_history_df.loc[spy.Value]  # or all_history_df.loc["SPY"]
```

After the index renaming, the unstacked `DataFrame` has the following format:

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[spy].EndTime)),
    new DecimalDataFrameColumn("SPY Open", history.Select(x => x[spy].Open)),
    new DecimalDataFrameColumn("SPY High", history.Select(x => x[spy].High)),
    new DecimalDataFrameColumn("SPY Low", history.Select(x => x[spy].Low)),
    new DecimalDataFrameColumn("SPY Close", history.Select(x => x[spy].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["SPY close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Equity subscriptions. To avoid issues, check if the `Slice` contains data for your Equity before you index it with the Equity `Symbol`.

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(spy))
    {
        var tradeBar = slice.Bars[spy];
    }
    if (slice.QuoteBars.ContainsKey(spy))
    {
        var quoteBar = slice.QuoteBars[spy];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(spy):
        trade_bar = slice.Bars[spy]
    if slice.QuoteBars.ContainsKey(spy):
        quote_bar = slice.QuoteBars[spy]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice`.

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
```

```
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol` .

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(spy)).Select(slice => slice.Bars[spy]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars` , iterate through the `TradeBars` to get the `TradeBar` of each Equity. The `TradeBars` may not have data for all of your Equity subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Equity `Symbol` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(spy))
    {
        var tradeBar = tradeBars[spy];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(spy):
        trade_bar = trade_bars[spy]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Equity. The `QuoteBars` may not have data for all of your Equity subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Equity `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(spy))
    {
        var quoteBar = quoteBars[spy];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(spy):
        quote_bar = quote_bars[spy]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Equity. The `Ticks` may not have data for all of your Equity subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Equity `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(spy))
    {
        var tick = ticks[spy];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(spy):
        ticks = ticks[spy]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some historical Equity data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(spy, datetime(2021, 11, 23), datetime(2021, 12, 8), Resolution.Daily).loc[spy]
   ```

   ```
   var history = qb.History<TradeBar>(spy, new DateTime(2021, 11, 23), new DateTime(2021, 12, 8), Resolution.Daily);
   ```

2. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

3. Create a `Candlestick` chart.

   ```
   candlestick = go.Candlestick(x=history.index,
                                open=history['open'],
                                high=history['high'],
                                low=history['low'],
                                close=history['close'])
   ```

   ```
   var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
       history.Select(x => x.Open),
       history.Select(x => x.High),
       history.Select(x => x.Low),
       history.Select(x => x.Close),
       history.Select(x => x.EndTime)
   );
   ```

4. Create a `Layout` .

   ```
   layout = go.Layout(title=go.layout.Title(text='SPY OHLC'),
                      xaxis_title='Date',
                      yaxis_title='Price',
                      xaxis_rangeslider_visible=False)
   ```

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Price ($)");
   Title title = Title.init("SPY Price");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

5. Create the `Figure` .
6. Assign the `Layout` to the chart.

   ```
   fig = go.Figure(data=[candlestick], layout=layout)
   ```

   ```
   chart.WithLayout(layout);
   ```

7. Show the plot.

   ```
   fig.show()
   ```

   ```
   HTML(GenericChart.toChartHTML(chart))
   ```

   Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

   ```
   history = qb.History([spy, tlt], datetime(2021, 11, 23), datetime(2021, 12, 8), Resolution.Daily)
   ```

   ```
   var history = qb.History<TradeBar>(new [] {spy, tlt}, new DateTime(2021, 11, 23), new DateTime(2021, 12, 8), Resolution.Daily);
   ```

2. Select the data to plot.

   ```
   volume = history['volume'].unstack(level=0)
   ```

   ```
   var spy = history.Select(x => x["SPY"]);
   ```

3. Call the `plot` method on the `pandas` object.
4. Create a `Line` chart.

   ```
   volume.plot(title="Volume", figsize=(15, 10))
   ```

   ```
   var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
       spy.Select(x => x.EndTime),
       spy.Select(x => x.Volume)
   );
   ```

5. Create a `Layout` .

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Volume");
   Title title = Title.init("SPY Volume");

   Layout layout = new Layout();
   ```

```
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

Common Errors

Some factor files have INF split values, which indicate that the stock has so many splits that prices can't be calculated with correct numerical precision. To allow history requests with these symbols, we need to move the starting date forward when reading the data. If there are numerical precision errors in the factor files for a security in your history request, LEAN throws the following error:

"Warning: when performing history requests, the start date will be adjusted if there are numerical precision errors in the factor files."

3.3 Equity Fundamental Data

Introduction

This page explains how to request, manipulate, and visualize historical Equity Fundamental data. Corporate fundamental data is available through the [US Fundamental Data from Morningstar](#) .

Create Subscriptions

Follow these steps to subscribe to an Equity security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
using QuantConnect.Indicators;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddEquity` method with a ticker and then save a reference to the Equity `Symbol` .

```
var symbols = new []
    {
        "AAL",  // American Airlines Group, Inc.
        "ALGT", // Allegiant Travel Company
        "ALK",  // Alaska Air Group, Inc.
        "DAL",  // Delta Air Lines, Inc.
        "LUV",  // Southwest Airlines Company
        "SKYW", // SkyWest, Inc.
        "UAL"   // United Air Lines
    }
    .Select(ticker => qb.AddEquity(ticker).Symbol);
```

```
symbols = [qb.AddEquity(ticker).Symbol
    for ticker in [
        "AAL",   # American Airlines Group, Inc.
        "ALGT",  # Allegiant Travel Company
        "ALK",   # Alaska Air Group, Inc.
        "DAL",   # Delta Air Lines, Inc.
        "LUV",   # Southwest Airlines Company
        "SKYW",  # SkyWest, Inc.
        "UAL"    # United Air Lines
    ]]
```

To view the supported assets in the US Equities dataset, see the [Data Explorer](#) .

Get Historical Data

You need a [subscription](#) before you can request historical fundamental data for a US Equity.

To get historical data, call the `GetFundamental` method with a list of `Symbol` objects, a fundamental data field name, a start `DateTime datetime` , and an end `DateTime datetime` . The start and end times you provide are based in the [notebook time zone](#) . To view the possible fundamental data field names, see the `FineFundamental` attributes in [Data Point Attributes](#) . For example, to get data for airline companies over 2014, run:

```
var startTime = new DateTime(2014, 1, 1);
var endTime = new DateTime(2015, 1, 1);
var history = qb.GetFundamental(symbols, "ValuationRatios.PERatio", startTime, endTime);
```

```
start_time = datetime(2014, 1, 1)
end_time = datetime(2015, 1, 1)
history = qb.GetFundamental(symbols, "ValuationRatios.PERatio", start_time, end_time)
```

The preceding method returns the fundamental data field values that are timestamped within the defined period of time.

Wrangle Data

You need some [historical data](#) to perform wrangling operations. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some [historical data](#) to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data.

The `DataFrame` index is the `EndTime` of the data sample. The columns of the `DataFrame` are the Equity `Symbol` objects.

To get the fundamental data points for each Equity, iterate through the history request result.

To select the historical data of a single Equity, index the `DataFrame` with the Equity `Symbol` . Each history slice may not have data for all of your Equity subscriptions. To avoid issues, check if it contains data for your Equity before you index it with the Equity `Symbol` .

```
foreach (var slice in history)
{
    foreach (var symbol in symbols)
    {
        if (slice.ContainsKey(symbol))
        {
            var peRatio = slice[symbol];
        }
    }
}
```

```
history[symbols[1]]
```

You can also iterate through each data point in the slice.

```
foreach (var slice in history)
{
    foreach (var kvp in slice)
    {
        var symbol = kvp.Key;
        var peRatio = kvp.Value;
    }
}
```

You can also use LINQ to select the historical data points.

```
var symbol = symbols.Last();
var values = history.Select(slice => slice[symbol]);
```

Plot Data

You need some [historical Equity fundamental data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot line charts.

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Call the `plot` method on the history `DataFrame` .

   ```
   history.plot(title='PE Ratio Over Time', figsize=(15, 8))
   ```

2. Create `Line` charts.

   ```
   var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
       history.Select(x => (DateTime)x.Time),
       history.Select(x => (decimal)x[aapl]),
       Name: "AAPL"
   );
   var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
       history.Select(x => (DateTime)x.Time),
       history.Select(x => (decimal)x[goog]),
       Name: "GOOG"
   );
   ```

3. Create a `Layout` .

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "PE Ratio");
   Title title = Title.init("AAPL & GOOG PE Ratio");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

4. Combine the charts and assign the `Layout` to the chart.

   ```
   var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
   chart.WithLayout(layout);
   ```

5. Show the plot.

   ```
   plt.show()
   ```

   ```
   HTML(GenericChart.toChartHTML(chart))
   ```

   Line charts display the value of the property you selected in a time series.

3.4 Equity Options

Introduction

This page explains how to request, manipulate, and visualize historical Equity Options data.

Create Subscriptions

Follow these steps to subscribe to an Equity Option security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Securities;
using QuantConnect.Securities.Option;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Subscribe to the underlying Equity with raw data normalization and save a reference to the Equity Symbol .

```
var equitySymbol = qb.AddEquity("SPY", dataNormalizationMode: DataNormalizationMode.Raw).Symbol;
```

```
equity_symbol = qb.AddEquity("SPY", dataNormalizationMode=DataNormalizationMode.Raw).Symbol
```

To view the supported underlying assets in the US Equity Options dataset, see the Data Explorer .

4. Call the `AddOption` method with the underlying Equity `Symbol` .

```
var option = qb.AddOption(equitySymbol);
```

```
option = qb.AddOption(equity_symbol)
```

5. (Optional) Set a contract filter .

```
option.SetFilter(-1, 1, 0, 90);
```

```
option.SetFilter(-1, 1, 0, 90)
```

The filter determines which contracts the `GetOptionHistory` method returns. If you don't set a filter, the default filter selects the contracts that have the following characteristics:

- Standard type (exclude weeklys)
- Within 1 strike price of the underlying asset price
- Expire within 31 days

- (Optional) Set the price model .

```
option.PriceModel = OptionPriceModels.BjerksundStensland();
```

```
option.PriceModel = OptionPriceModels.BjerksundStensland()
```

If you want historical data on individual contracts and their `OpenInterest` , follow these steps to subscribe to the individual Equity Option contracts:

1. Call the `GetOptionsContractList` method with the underlying `Equity Symbol` and a `datetime DateTime` .

```
var startDate = new DateTime(2021, 12, 31);
var contractSymbols = qb.OptionChainProvider.GetOptionContractList(equitySymbol, startDate);
```

```
start_date = datetime(2021, 12, 31)
contract_symbols = qb.OptionChainProvider.GetOptionContractList(equity_symbol, start_date)
```

This method returns a list of `Symbol` objects that reference the Option contracts that were trading at the given time. If you set a contract filter with `SetFilter` , it doesn't affect the results of `GetOptionsContractList` .

2. Select the `Symbol` of the `OptionContract` object(s) for which you want to get historical data.

To filter and select contracts, you can use the following properties of each `Symbol` object:

Property	Description
`ID.Date`	The expiration date of the contract.
`ID.StrikePrice`	The strike price of the contract.
`ID.OptionRight`	The contract type. The `OptionRight` enumeration has the following members:
`ID.OptionStyle`	The contract style. The `OptionStyle` enumeration has the following members:

```
var contractSymbol = contractSymbols.Where(s =>
    s.ID.OptionRight == OptionRight.Call &&
    s.ID.StrikePrice == 477 &&
    s.ID.Date == new DateTime(2022, 1, 21)).First();
```

```
contract_symbol = [s for s in contract_symbols
    if s.ID.OptionRight == OptionRight.Call
        and s.ID.StrikePrice == 477
        and s.ID.Date == datetime(2022, 1, 21)][0]
```

3. Call the `AddOptionContract` method with an `OptionContract Symbol` and disable fill-forward.

```
var optionContract = qb.AddOptionContract(contractSymbol, fillForward: false);
```

```
option_contract = qb.AddOptionContract(contract_symbol, fillForward = False)
```

Disable fill-forward because there are only a few `OpenInterest` data points per day.

4. (Optional) Set the price model .

```
optionContract.PriceModel = OptionPriceModels.BjerksundStensland();
```

```
option_contract.PriceModel = OptionPriceModels.BjerksundStensland()
```

Get Historical Data

You need a subscription before you can request historical data for Equity Option contracts. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the contract dimension, you can request historical data for a single contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Before you request historical data, call the `SetStartDate` method with a `datetime DateTime` to reduce the risk of look-ahead bias .

```
qb.SetStartDate(startDate);
```

```
qb.SetStartDate(start_date)
```

If you call the `SetStartDate` method, the date that you pass to the method is the latest date for which your history requests will return data.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the contract `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(contractSymbol, 10);
var subsetHistorySlice = qb.History(new[] {contractSymbol}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, 400);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, 400);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, 400);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, 10)
subset_history_df = qb.History([contract_symbol], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, 400)
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], 400)
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, 400)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, 10)
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, 10)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, 400)
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], 400)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, 400)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the contract `Symbol` object(s) and a `TimeSpan timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(contractSymbol, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {contractSymbol}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, TimeSpan.FromDays(2));
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, TimeSpan.FromDays(2));
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, TimeSpan.FromDays(2));
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, timedelta(days=3))
subset_history_df = qb.History([contract_symbol], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], timedelta(days=3))
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, timedelta(days=3))
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], timedelta(days=3))
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, timedelta(days=3))

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, timedelta(days=2))
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], timedelta(days=2))
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, timedelta(days=2))
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for individual Equity Option contracts during a specific period of time, call the `History` method with the Equity Option contract `Symbol` object(s), a start `DateTime datetime` , and an end `DateTime datetime` . The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2021, 12, 1);
var endTime = new DateTime(2021, 12, 31);

// Slice objects
var singleHistorySlice = qb.History(contractSymbol, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {contractSymbol}, startTime, endTime);
var allHistorySlice = qb.History(startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, startTime, endTime);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, startTime, endTime);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, startTime, endTime);
```

```
start_time = datetime(2021, 12, 1)
end_time = datetime(2021, 12, 31)

# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, start_time, end_time)
subset_history_df = qb.History([contract_symbol], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, start_time, end_time)
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], start_time, end_time)
all_history_trade_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, start_time, end_time)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, start_time, end_time)
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], start_time, end_time)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, start_time, end_time)
```

To get historical data for all of the Equity Option contracts that pass your [filter](#) during a specific period of time, call the `GetOptionHistory` method with the underlying Equity `Symbol` object, a start `DateTime datetime` , and an end `DateTime datetime` .

```
option_history = qb.GetOptionHistory(equity_symbol, end_time-timedelta(days=2), end_time, Resolution.Minute, fillForward=False, extendedMarketHours=False)
```

```
var optionHistory = qb.GetOptionHistory(equitySymbol, endTime-TimeSpan.FromDays(2), endTime, Resolution.Minute, fillForward: False, extendedMarketHours: False);
```

The preceding calls return data that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Equity Option contract subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`				
`Second`				
`Minute`	☐	☐		
`Hour`	☐	☐		
`Daily`	☐	☐		

Markets

LEAN groups all of the US Equity Option exchanges under `Market.USA` , so you don't need to pass a `Market` to the `AddOption` or `AddOptionContract` methods.

Wrangle Data

You need some [historical data](#) to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some [historical data](#) to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If your history request returns a `DataFrame` , the `DataFrame` has the following index levels:

1. Contract expiry
2. Contract strike price
3. Contract type (call or put)
4. [Encoded contract Symbol](#)
5. The `EndTime` of the data sample

The columns of the `DataFrame` are the data properties. Depending on how you request data, the `DataFrame` may contain data for the underlying security, which causes some of the index levels to be an empty string for the corresponding rows.

To select the rows of the contract(s) that expire at a specific time, index the `loc` property of the `DataFrame` with the expiry time.

```
all_history_df.loc[datetime(2022, 1, 21)]
```

If you remove the first three index levels, you can index the `DataFrame` with just the contract `Symbol` , similiar to how you would with non-derivative asset classes. To remove the first three index levels, call the droplevel method.

```
all_history_df.index = all_history_df.index.droplevel([0,1,2])
```

To select the historical data of a single Equity Options contract, index the `loc` property of the `DataFrame` with the contract `Symbol` .

```
all_history_df.loc[contract_symbol]
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[contract_symbol]['close']
```

If you request historical data for multiple Equity Option contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the Equity Option contracts. To transform the `DataFrame` , select the column you want to display for each Equity Option contract and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each security and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[contractSymbol].EndTime)),
    new DecimalDataFrameColumn(" Open", history.Select(x => x[contractSymbol].Open)),
    new DecimalDataFrameColumn(" High", history.Select(x => x[contractSymbol].High)),
    new DecimalDataFrameColumn(" Low", history.Select(x => x[contractSymbol].Low)),
    new DecimalDataFrameColumn(" Close", history.Select(x => x[contractSymbol].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df[" close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Equity Options subscriptions. To avoid issues, check if the `Slice` contains data for your Equity Option contract before you index it with the Equity Options `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(contractSymbol))
    {
        var tradeBar = slice.Bars[contractSymbol];
    }
    if (slice.QuoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = slice.QuoteBars[contractSymbol];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(contract_symbol):
        trade_bar = slice.Bars[contract_symbol]
    if slice.QuoteBars.ContainsKey(contract_symbol):
        quote_bar = slice.QuoteBars[contract_symbol]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol` .

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(contractSymbol)).Select(slice => slice.Bars[contractSymbol]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars` , iterate through the `TradeBars` to get the `TradeBar` of each Equity Option contract. The `TradeBars` may not have data for all of your Equity Options subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Equity Options `Symbol` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(contractSymbol))
    {
        var tradeBar = tradeBars[contractSymbol];
    }
}
```

```
for trade_bars in all_history_trade_bars:
```

```
    if trade_bars.ContainsKey(contract_symbol):
        trade_bar = trade_bars[contract_symbol]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Equity Option contract. The `QuoteBars` may not have data for all of your Equity Options subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Equity Options `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = quoteBars[contractSymbol];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(contract_symbol):
        quote_bar = quote_bars[contract_symbol]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

OpenInterest Objects

If the `History` method returns `OpenInterest` objects, iterate through the `OpenInterest` objects to get each one.

```
foreach (var openInterest in singleHistoryOpenInterest)
{
    Console.WriteLine(openInterest);
}
```

```
for open_interest in single_history_open_interest:
    print(open_interest)
```

If the `History` method returns a dictionary of `OpenInterest` objects, iterate through the dictionary to get the `OpenInterest` of each Equity Option contract. The dictionary of `OpenInterest` objects may not have data for all of your Equity Options contract subscriptions. To avoid issues, check if the dictionary contains data for your contract before you index it with the Equity Options contract `Symbol` .

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    if (openInterestDict.ContainsKey(contractSymbol))
    {
        var openInterest = openInterestDict[contractSymbol];
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    if open_interest_dict.ContainsKey(contract_symbol):
        open_interest = open_interest_dict[contract_symbol]
```

You can also iterate through each of the `OpenInterest` dictionaries.

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    foreach (var kvp in openInterestDict)
    {
        var symbol = kvp.Key;
        var openInterest = kvp.Value;
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    for kvp in open_interest_dict:
        symbol = kvp.Key
        open_interest = kvp.Value
```

OptionHistory Objects

The `GetOptionHistory` method returns an `OptionHistory` object. To get each [slice](#) in the `OptionHistory` object, iterate through it.

```
foreach (var slice in optionHistory)
{
    foreach (var kvp in slice.OptionChains)
    {
        var canonicalSymbol = kvp.Key;
        var chain = kvp.Value;
        foreach (var contract in chain)
        {

        }
    }
}
```

```
for slice in option_history:
    for canonical_symbol, chain in slice.OptionChains.items():
        for contract in chain:
            pass
```

To convert the `OptionHistory` object to a `DataFrame` that contains the trade and quote information of each contract and the underlying, call the `GetAllData` method.

```
option_history.GetAllData()
```

To get the expiration dates of all the contracts in an `OptionHistory` object, call the `GetExpiryDates` method.

```
option_history.GetExpiryDates()
```

To get the strike prices of all the contracts in an `OptionHistory` object, call the `GetStrikes` method.

```
option_history.GetStrikes()
```

Plot Data

You need some [historical Equity Options data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

```
history = qb.History(contract_symbol, datetime(2021, 12, 30), datetime(2021, 12, 31))
```

```
var history = qb.History<TradeBar>(contractSymbol, new DateTime(2021, 12, 30), new DateTime(2021, 12, 31));
```

2. Drop the first four index levels of the `DataFrame` that returns.

```
history.index = history.index.droplevel([0,1,2,3])
```

3. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

4. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

5. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text=f'{symbol.Value} OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{contractSymbol} Price");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

7. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

8. Show the plot.

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

The Jupyter Notebook displays a candlestick chart of the Option contract's price.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History(OpenInterest, contract_symbol, datetime(2021, 12, 1), datetime(2021, 12, 31))
```

```
var history = qb.History<OpenInterest>(contract_symbol, new DateTime(2021, 12, 1), new DateTime(2021, 12, 31));
```

2. Drop the first three index levels of the `DataFrame` that returns.

```
history.index = history.index.droplevel([0, 1, 2])
```

3. Select the open interest data.

```
history = history['openinterest'].unstack(level=0)
```

4. Rename the column to the `Symbol` of the contract.

```
history.columns = [
    Symbol.GetAlias(SecurityIdentifier.Parse(x), equity_symbol)
        for x in history.columns]
```

5. Call the `plot` method with a title.

```
history.plot(title="Open Interest")
```

6. Create a `Line` chart.

```
var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x.EndTime),
    history.Select(x => x.Value)
);
```

7. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
```

```
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Open Interest");
Title title = Title.init("SPY Open Interest");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

8. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

9. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

The Jupyter Notebook displays a line chart of open interest data.

Get Price Model Data

Follow these steps to get the values of theoretical prices, implied volatility, and Greeks:

1. Create subscriptions and set the price model .
2. Set the underlying volatility model .

```
qb.Securities[equity_symbol].VolatilityModel = StandardDeviationOfReturnsVolatilityModel(30, Resolution.Daily)
```

You need to reset the volatility before you start calculating the theoretical prices, implied volatility, and Greeks.

3. Get historical data for the underlying Equity and the Option contract(s).
4. Iterate through the historical data and calculate the values.

```
df = pd.DataFrame()
for slice in history:
    underlying_price = None
    underlying_volatility = None

    # Update the security with QuoteBar information
    for bar in slice.QuoteBars.Values:
        qb.Securities[bar.Symbol].SetMarketPrice(bar)

    # Update the security with TradeBar information
    for bar in slice.Bars.Values:
        symbol = bar.Symbol
        security = qb.Securities[symbol]
        security.SetMarketPrice(bar)

        if security.Type == SecurityType.Equity:
            underlying_volatility = security.VolatilityModel.Volatility
            underlying_price = security.Price
            continue

        # Create the Option contract
        contract = OptionContract.Create(symbol, symbol.Underlying, bar.EndTime, security, underlying_price)
        contract.LastPrice = bar.Close

        # Evaluate the price model to get the IV, Greeks, and theoretical price
        result = security.PriceModel.Evaluate(security, None, contract)
        greeks = result.Greeks

        # Append the data to the DataFrame
        data = {
            "IV" : result.ImpliedVolatility,
            "Delta": greeks.Delta,
            "Gamma": greeks.Gamma,
            "Vega": greeks.Vega,
            "Rho": greeks.Rho,
            "Theta": greeks.Theta,
            "LastPrice": contract.LastPrice,
            "Close": security.Close,
            "theoreticalPrice" : result.TheoreticalPrice,
            "underlyingPrice": underlying_price,
            "underlyingVolatility": underlying_volatility
        }
        right = "Put" if symbol.ID.OptionRight == 1 else "Call"
        index = pd.MultiIndex.from_tuples([(symbol.ID.Date, symbol.ID.StrikePrice, right, symbol.Value, bar.EndTime)], names=["expiry", "strike", "type", "symbol", "endTime"
        df = pd.concat([df, pd.DataFrame(data, index=index)])
```

For a full example, see the following project:

- Charts
- Statistics
- Code
    - main.py
    - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	0
Average Win	0%
Average Loss	0%
Compounding Annual Return	0%
Drawdown	0%
Expectancy	0
Net Profit	0%
Sharpe Ratio	0
Probabilistic Sharpe Ratio	0%
Loss Rate	0%
Win Rate	0%
Profit-Loss Ratio	0
Alpha	0
Beta	0
Annual Standard Deviation	0
Annual Variance	0
Information Ratio	0.92
Tracking Error	0.233
Treynor Ratio	0
Total Fees	$0.00
Estimated Strategy Capacity	$0
Lowest Capacity Asset	

```
# region imports
from AlgorithmImports import *
# endregion

class FormalBlackAnguilline(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2022, 9, 3)
        self.SetEndDate(2022, 10, 4)
        self.SetCash(100000)  # Set Strategy Cash
```

3.5 Crypto

Introduction

This page explains how to request, manipulate, and visualize historical Crypto data.

Create Subscriptions

Follow these steps to subscribe to a Crypto security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddCrypto` method with a ticker and then save a reference to the Crypto `Symbol` .

```
var btcusd = qb.AddCrypto("BTCUSD").Symbol;
var ethusd = qb.AddCrypto("ETHUSD").Symbol;
```

```
btcusd = qb.AddCrypto("BTCUSD").Symbol
ethusd = qb.AddCrypto("ETHUSD").Symbol
```

To view the supported assets in the Crypto datasets, see the Supported Assets section of the CoinAPI dataset listings .

Get Historical Data

You need a subscription before you can request historical data for a security. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the security dimension, you can request historical data for a single Cryptocurrency, a subset of the Cryptocurrencies you created subscriptions for in your notebook, or all of the Cryptocurrencies in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(btcusd, 10);
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, 10)
subset_history_df = qb.History([btcusd, ethusd], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, 10)
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, 10)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)
```

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(btcusd, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(btcusd, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, timedelta(days=3))
subset_history_df = qb.History([btcusd, ethusd], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], timedelta(days=3))
```

```
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(btcusd, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([btcusd, ethusd], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](btcusd, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([btcusd, ethusd], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 2, 1);

// Slice objects
var singleHistorySlice = qb.History(btcusd, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(btcusd, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {btcusd, ethusd}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, start_time, end_time)
subset_history_df = qb.History([btcusd, ethusd], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(btcusd, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([btcusd, ethusd], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](btcusd, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([btcusd, ethusd], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

Resolutions

The following table shows the available resolutions and data formats for Crypto subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`			☐	☐
`Second`	☐	☐		
`Minute`	☐	☐		
`Hour`	☐	☐		
`Daily`	☐	☐		

Markets

The following `Market` enumeration members are available for Crypto:

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded Crypto Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single Crypto, index the `loc` property of the `DataFrame` with the Crypto `Symbol` .

```
all_history_df.loc[btcusd]  # or all_history_df.loc['BTCUSD']
```

To select a column of the `DataFrame`, index it with the column name.

```
all_history_df.loc[btcusd]['close']
```

If you request historical data for multiple Crypto pairs, you can transform the `DataFrame` so that it's a time series of close values for all of the Crypto pairs. To transform the `DataFrame`, select the column you want to display for each Crypto pair and then call the [unstack](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.unstack.html) method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each Crypto pair and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[btcusd].EndTime)),
    new DecimalDataFrameColumn("BTCUSD Open", history.Select(x => x[btcusd].Open)),
    new DecimalDataFrameColumn("BTCUSD High", history.Select(x => x[btcusd].High)),
    new DecimalDataFrameColumn("BTCUSD Low", history.Select(x => x[btcusd].Low)),
    new DecimalDataFrameColumn("BTCUSD Close", history.Select(x => x[btcusd].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["BTCUSD close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Crypto subscriptions. To avoid issues, check if the `Slice` contains data for your Crypto pair before you index it with the Crypto `Symbol`.

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(btcusd))
    {
        var tradeBar = slice.Bars[btcusd];
    }
    if (slice.QuoteBars.ContainsKey(btcusd))
    {
        var quoteBar = slice.QuoteBars[btcusd];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(btcusd):
        trade_bar = slice.Bars[btcusd]
    if slice.QuoteBars.ContainsKey(btcusd):
        quote_bar = slice.QuoteBars[btcusd]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice`.

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol`.

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(btcusd)).Select(slice => slice.Bars[btcusd]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars`, iterate through the `TradeBars` to get the `TradeBar` of each Crypto pair. The `TradeBars` may not have data for all of your Crypto subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Crypto `Symbol`.

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(btcusd))
    {
        var tradeBar = tradeBars[btcusd];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(btcusd):
        trade_bar = trade_bars[btcusd]
```

You can also iterate through each of the `TradeBars`.

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
```

```
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Crypto pair. The `QuoteBars` may not have data for all of your Crypto subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Crypto `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(btcusd))
    {
        var quoteBar = quoteBars[btcusd];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(btcusd):
        quote_bar = quote_bars[btcusd]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Crypto pair. The `Ticks` may not have data for all of your Crypto subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Crypto `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(btcusd))
    {
        var tick = ticks[btcusd];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(btcusd):
        ticks = ticks[btcusd]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some historical Crypto data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(btcusd, datetime(2020, 12, 27), datetime(2021, 12, 21), Resolution.Daily).loc[btcusd]
   ```

   ```
   var history = qb.History<TradeBar>(btcusd, new DateTime(2020, 12, 27), new DateTime(2021, 12, 21), Resolution.Daily);
   ```

2. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

3. Create a `Candlestick` .

   ```
   candlestick = go.Candlestick(x=history.index,
                                 open=history['open'],
                                 high=history['high'],
                                 low=history['low'],
                                 close=history['close'])
   ```

   ```
   var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
       history.Select(x => x.Open),
       history.Select(x => x.High),
   ```

```
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='BTCUSD OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{btcusd} Price");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([btcusd, ethusd], datetime(2020, 12, 27), datetime(2021, 12, 21), Resolution.Daily)
```

```
var history = qb.History<TradeBar>(new[] {btcusd, ethusd}, new DateTime(2020, 12, 27), new DateTime(2021, 12, 21), Resolution.Daily);
```

2. Select the data to plot.

```
volume = history['volume'].unstack(level=0)
```

3. Call the `plot` method on the `pandas` object.

```
volume.plot(title="Volume", figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[btcusd].EndTime),
    history.Select(x => x[btcusd].Volume),
    Name: "BTCUSD"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[ethusd].EndTime),
    history.Select(x => x[ethusd].Volume),
    Name: "ETHUSD"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Volume");
Title title = Title.init("BTCUSD & ETHUSD Volume");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.6 Crypto Futures

Introduction

This page explains how to request, manipulate, and visualize historical Crypto Futures data.

Create Subscriptions

Follow these steps to subscribe to a perpetual Crypto Futures contract:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddCryptoFuture` method with a ticker and then save a reference to the Crypto Future `Symbol` .

```
var btcusd = qb.AddCryptoFuture("BTCUSD").Symbol;
var ethusd = qb.AddCryptoFuture("ETHUSD").Symbol;
```

```
btcusd = qb.AddCryptoFuture("BTCUSD").Symbol
ethusd = qb.AddCryptoFuture("ETHUSD").Symbol
```

To view the supported assets in the Crypto Futures datasets, see the [Data Explorer](#) .

Get Historical Data

You need a [subscription](#) before you can request historical data for a security. You can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. You can also request historical data for a single contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(btcusd, 10);
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, 10)
subset_history_df = qb.History([btcusd, ethusd], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, 10)
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, 10)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)
```

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(btcusd, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(btcusd, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {btcusd, ethusd}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, timedelta(days=3))
subset_history_df = qb.History([btcusd, ethusd], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], timedelta(days=3))
```

```
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(btcusd, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([btcusd, ethusd], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](btcusd, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([btcusd, ethusd], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 2, 1);

// Slice objects
var singleHistorySlice = qb.History(btcusd, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {btcusd, ethusd}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(btcusd, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {btcusd, ethusd}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(btcusd, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {btcusd, ethusd}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(btcusd, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {btcusd, ethusd}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of trade and quote data
single_history_df = qb.History(btcusd, start_time, end_time)
subset_history_df = qb.History([btcusd, ethusd], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, btcusd, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [btcusd, ethusd], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, btcusd, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [btcusd, ethusd], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(btcusd, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([btcusd, ethusd], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](btcusd, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([btcusd, ethusd], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](btcusd, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([btcusd, ethusd], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](btcusd, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([btcusd, ethusd], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

Resolutions

The following table shows the available resolutions and data formats for Crypto Futures contract subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`			☐	☐
`Second`	☐	☐		
`Minute`	☐	☐		
`Hour`	☐	☐		
`Daily`	☐	☐		

Markets

Crypto Futures are currently only available on `Market.Binance` .

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded Crypto Future Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single Crypto Future, index the `loc` property of the `DataFrame` with the Crypto Future `Symbol`.

```
all_history_df.loc[btcusd]  # or all_history_df.loc['BTCUSD']
```

To select a column of the `DataFrame`, index it with the column name.

```
all_history_df.loc[btcusd]['close']
```

If you request historical data for multiple Crypto Futures contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the Crypto Futures contracts. To transform the `DataFrame`, select the column you want to display for each Crypto Futures contract and then call the [unstack](#) method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each Crypto Futures contract and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[btcusd].EndTime)),
    new DecimalDataFrameColumn("BTCUSD Open", history.Select(x => x[btcusd].Open)),
    new DecimalDataFrameColumn("BTCUSD High", history.Select(x => x[btcusd].High)),
    new DecimalDataFrameColumn("BTCUSD Low", history.Select(x => x[btcusd].Low)),
    new DecimalDataFrameColumn("BTCUSD Close", history.Select(x => x[btcusd].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["BTCUSD close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Crypto Future subscriptions. To avoid issues, check if the `Slice` contains data for your Crypto Futures contract before you index it with the Crypto Future `Symbol`.

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(btcusd))
    {
        var tradeBar = slice.Bars[btcusd];
    }
    if (slice.QuoteBars.ContainsKey(btcusd))
    {
        var quoteBar = slice.QuoteBars[btcusd];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(btcusd):
        trade_bar = slice.Bars[btcusd]
    if slice.QuoteBars.ContainsKey(btcusd):
        quote_bar = slice.QuoteBars[btcusd]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice`.

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol`.

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(btcusd)).Select(slice => slice.Bars[btcusd]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars`, iterate through the `TradeBars` to get the `TradeBar` of each Crypto Futures contract. The `TradeBars` may not have data for all of your Crypto Future subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Crypto Future `Symbol`.

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(btcusd))
    {
        var tradeBar = tradeBars[btcusd];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(btcusd):
        trade_bar = trade_bars[btcusd]
```

You can also iterate through each of the `TradeBars`.

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
```

```
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Crypto Futures contract. The `QuoteBars` may not have data for all of your Crypto Future subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Crypto Future `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(btcusd))
    {
        var quoteBar = quoteBars[btcusd];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(btcusd):
        quote_bar = quote_bars[btcusd]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Crypto Futures contract. The `Ticks` may not have data for all of your Crypto Future subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Crypto Future `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(btcusd))
    {
        var tick = ticks[btcusd];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(btcusd):
        ticks = ticks[btcusd]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some historical Crypto Futures data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

```
history = qb.History(btcusd, datetime(2021, 11, 23), datetime(2021, 12, 8), Resolution.Daily).loc[btcusd]
```

```
var history = qb.History<TradeBar>(btcusd, new DateTime(2021, 11, 23), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='BTCUSD 18R OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init("BTCUSD 18R OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([btcusd, ethusd], datetime(2021, 11, 23), datetime(2021, 12, 8), Resolution.Daily)
```

2. Select the data to plot.

```
volume = history['volume'].unstack(level=0)
```

3. Call the `plot` method on the `pandas` object.

```
volume.plot(title="Volume", figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[btcusd].EndTime),
    history.Select(x => x[btcusd].Volume),
    Name: "BTCUSD 18R"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[ethusd].EndTime),
    history.Select(x => x[ethusd].Volume),
    Name: "ETHUSD 18R"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Volume");
Title title = Title.init("BTCUSD 18R & ETHUSD 18R Volume");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.7 Futures

Introduction

This page explains how to request, manipulate, and visualize historical Futures data.

Create Subscriptions

Follow these steps to subscribe to a Future security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Securities;
using QuantConnect.Securities.Future;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddFuture` method with a ticker, resolution, and [contract rollover settings](#) .

```
var future = qb.AddFuture(Futures.Indices.SP500EMini, Resolution.Minute,
                dataNormalizationMode: DataNormalizationMode.BackwardsRatio,
                dataMappingMode: DataMappingMode.LastTradingDay,
                contractDepthOffset: 0);
```

```
future = qb.AddFuture(Futures.Indices.SP500EMini, Resolution.Minute,
                dataNormalizationMode = DataNormalizationMode.BackwardsRatio,
                dataMappingMode = DataMappingMode.LastTradingDay,
                contractDepthOffset = 0)
```

To view the available tickers in the US Futures dataset, see [Supported Assets](#) .

If you omit any of the arguments after the ticker, see the following table for their default values:

Argument	Default Value
`resolution`	`Resolution.Minute`
`dataNormalizationMode`	`DataNormalizationMode.Adjusted`
`dataMappingMode`	`DataMappingMode.OpenInterest`
`contractDepthOffset`	0

4. *(Optional)* Set a [contract filter](#) .

```
future.SetFilter(0, 90);
```

```
future.SetFilter(0, 90)
```

If you don't call the `SetFilter` method, the `GetFutureHistory` method won't return historical data.

If you want historical data on individual contracts and their `OpenInterest` , follow these steps to subscribe to individual Future contracts:

1. Call the `GetFuturesContractList` method with the underlying `Future Symbol` and a `datetime DateTime` .

```
var startDate = new DateTime(2021,12,20);
var symbols = qb.FutureChainProvider.GetFutureContractList(future.Symbol, startDate);
```

```
start_date = datetime(2021,12,20)
symbols = qb.FutureChainProvider.GetFutureContractList(future.Symbol, start_date)
```

This method returns a list of `Symbol` objects that reference the Future contracts that were trading at the given time. If you set a contract filter with `SetFilter` , it doesn't affect the results of `GetFutureContractList` .

2. Select the `Symbol` of the `FutureContract` object(s) for which you want to get historical data.

For example, select the `Symbol` of the contract with the closest expiry.

```
var contractSymbol = symbols.OrderBy(s => s.ID.Date).FirstOrDefault();
```

```
contract_symbol = sorted(symbols, key=lambda s: s.ID.Date)[0]
```

3. Call the `AddFutureContract` method with an `FutureContract Symbol` and disable fill-forward.

```
qb.AddFutureContract(contractSymbol, fillForward: false);
```

```
qb.AddFutureContract(contract_symbol, fillForward = False)
```

Disable fill-forward because there are only a few `OpenInterest` data points per day.

Get Historical Data

You need a [subscription](#) before you can request historical data for Futures contracts. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the contract dimension, you can request historical data for a single contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Before you request historical data, call the `SetStartDate` method with a `datetime DateTime` to reduce the risk of [look-ahead bias](#) .

```
qb.SetStartDate(startDate);
```

```
qb.SetStartDate(start_date)
```

If you call the `SetStartDate` method, the date that you pass to the method is the latest date for which your history requests will return data.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the contract `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(contractSymbol, 10);
var subsetHistorySlice = qb.History(new[] {contractSymbol}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, 400);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, 400);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, 400);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, 10)
subset_history_df = qb.History([contract_symbol], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, 400)
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], 400)
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, 400)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, 10)
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, 10)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, 400)
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], 400)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, 400)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

To get historical data for the continous Futures contract, in the preceding history requests, replace `contract_symbol` `contractSymbol` with `future.Symbol` .

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the contract `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(contractSymbol, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {contractSymbol}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History(contractSymbol, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History(new[] {contractSymbol}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, TimeSpan.FromDays(2));
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, TimeSpan.FromDays(2));
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, TimeSpan.FromDays(2));
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, timedelta(days=3))
subset_history_df = qb.History([contract_symbol], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], timedelta(days=3))
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, timedelta(days=3))
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], timedelta(days=3))
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, timedelta(days=3))

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](contract_symbol, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([contract_symbol], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, timedelta(days=2))
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], timedelta(days=2))
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, timedelta(days=2))
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

To get historical data for the continous Futures contract, in the preceding history requests, replace `contract_symbol` `contractSymbol` with `future.Symbol` .

Defined Period of Time

To get historical data for individual Futures contracts during a specific period of time, call the `History` method with the Futures contract `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2021, 12, 1);
var endTime = new DateTime(2021, 12, 31);

// Slice objects
```

```
var singleHistorySlice = qb.History(contractSymbol, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {contractSymbol}, startTime, endTime);
var allHistorySlice = qb.History(startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(contractSymbol, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {contractSymbol}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(contractSymbol, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {contractSymbol}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History(contractSymbol, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History(new[] {contractSymbol}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History(qb.Securities.Keys, startTime, endTime, Resolution.Tick);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(contractSymbol, startTime, endTime);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {contractSymbol}, startTime, endTime);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, startTime, endTime);
```

```
start_time = datetime(2021, 12, 1)
end_time = datetime(2021, 12, 31)

# DataFrame of trade and quote data
single_history_df = qb.History(contract_symbol, start_time, end_time)
subset_history_df = qb.History([contract_symbol], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, contract_symbol, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [contract_symbol], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, contract_symbol, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [contract_symbol], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, contract_symbol, start_time, end_time)
subset_history_open_interest_df = qb.History(OpenInterest, [contract_symbol], start_time, end_time)
all_history_trade_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, start_time, end_time)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](contract_symbol, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([contract_symbol], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](contract_symbol, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([contract_symbol], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](contract_symbol, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([contract_symbol], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](contract_symbol, start_time, end_time)
subset_history_open_interest = qb.History[OpenInterest]([contract_symbol], start_time, end_time)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, start_time, end_time)
```

To get historical data for the continous Futures contract, in the preceding history requests, replace `contract_symbol` `contractSymbol` with `future.Symbol` .

To get historical data for all of the Futures contracts that pass your filter during a specific period of time, call the `GetFutureHistory` method with the `Symbol` object of the continuous Future, a start `DateTime` `datetime` , and an end `DateTime` `datetime` .

```
future_history = qb.GetFutureHistory(future.Symbol, end_time-timedelta(days=2), end_time, Resolution.Minute, fillForward=False, extendedMarketHours=False)
```

```
var futureHistory = qb.GetFutureHistory(future.Symbol, endTime-TimeSpan.FromDays(2), endTime, Resolution.Minute, fillForward: False, extendedMarketHours: False);
```

The preceding calls return data that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Futures subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`				
`Second`				
`Minute`				
`Hour`				
`Daily`				

Markets

The following `Market` enumeration members are available for Futures:

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If your history request returns a `DataFrame` , the `DataFrame` has the following index levels:

1. Contract expiry
2. Encoded contract Symbol
3. The `EndTime` of the data sample

The columns of the `DataFrame` are the data properties. Depending on how you request data, the `DataFrame` may contain data for the continuous Futures contract. The continuous contract doesn't expire, so the default expiry date of December 30, 1899 doesn't have any practical meaning.

To select the rows of the contract(s) that expire at a specific time, index the `loc` property of the `DataFrame` with the expiry time.

```
all_history_df.loc[datetime(2022, 3, 18, 13, 30)]
```

If you remove the first index level, you can index the `DataFrame` with just the contract `Symbol`, similiar to how you would with non-derivative asset classes. To remove the first index level, call the [droplevel](#) method.

```
all_history_df.index = all_history_df.index.droplevel(0)
```

To select the historical data of a single Futures contract, index the `loc` property of the `DataFrame` with the contract `Symbol`.

```
all_history_df.loc[contract_symbol]
```

To select a column of the `DataFrame`, index it with the column name.

```
all_history_df.loc[contract_symbol]['close']
```

If you request historical data for multiple Futures contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the Futures contracts. To transform the `DataFrame`, select the column you want to display for each Futures contract and then call the [unstack](#) method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each security and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[contractSymbol].EndTime)),
    new DecimalDataFrameColumn(" Open", history.Select(x => x[contractSymbol].Open)),
    new DecimalDataFrameColumn(" High", history.Select(x => x[contractSymbol].High)),
    new DecimalDataFrameColumn(" Low", history.Select(x => x[contractSymbol].Low)),
    new DecimalDataFrameColumn(" Close", history.Select(x => x[contractSymbol].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df[" close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Futures subscriptions. To avoid issues, check if the `Slice` contains data for your Futures contract before you index it with the Futures `Symbol`.

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(contractSymbol))
    {
        var tradeBar = slice.Bars[contractSymbol];
    }
    if (slice.QuoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = slice.QuoteBars[contractSymbol];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(contract_symbol):
        trade_bar = slice.Bars[contract_symbol]
    if slice.QuoteBars.ContainsKey(contract_symbol):
        quote_bar = slice.QuoteBars[contract_symbol]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice`.

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol`.

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(contractSymbol)).Select(slice => slice.Bars[contractSymbol]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars`, iterate through the `TradeBars` to get the `TradeBar` of each Futures contract. The `TradeBars` may not have data for all of your Futures subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Futures `Symbol`.

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(contractSymbol))
    {
        var tradeBar = tradeBars[contractSymbol];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(contract_symbol):
        trade_bar = trade_bars[contract_symbol]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Futures contract. The `QuoteBars` may not have data for all of your Futures subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Futures `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = quoteBars[contractSymbol];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(contract_symbol):
        quote_bar = quote_bars[contract_symbol]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Futures contract. The `Ticks` may not have data for all of your Futures subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Futures `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(contractSymbol))
    {
        var tick = ticks[contractSymbol];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(contract_symbol):
        ticks = ticks[contract_symbol]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

OpenInterest Objects

If the `History` method returns `OpenInterest` objects, iterate through the `OpenInterest` objects to get each one.

```
foreach (var openInterest in singleHistoryOpenInterest)
{
    Console.WriteLine(openInterest);
}
```

```
for open_interest in single_history_open_interest:
    print(open_interest)
```

If the `History` method returns a dictionary of `OpenInterest` objects, iterate through the dictionary to get the `OpenInterest` of each Futures contract. The dictionary of `OpenInterest` objects may not have data for all of your Futures contract subscriptions. To avoid issues, check if the dictionary contains data for your contract before you index it with the Futures contract `Symbol` .

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    if (openInterestDict.ContainsKey(contractSymbol))
    {
        var openInterest = openInterestDict[contractSymbol];
    }
}
```

```
for open_interest_dict in all_history_open_interest:
```

```
    if open_interest_dict.ContainsKey(contract_symbol):
        open_interest = open_interest_dict[contract_symbol]
```

You can also iterate through each of the `OpenInterest` dictionaries.

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    foreach (var kvp in openInterestDict)
    {
        var symbol = kvp.Key;
        var openInterest = kvp.Value;
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    for kvp in open_interest_dict:
        symbol = kvp.Key
        open_interest = kvp.Value
```

FutureHistory Objects

The `GetFutureHistory` method returns a `FutureHistory` object. To get each [slice](#) in the `FutureHistory` object, iterate through it.

```
foreach (var slice in futureHistory)
{
    foreach (var kvp in slice.FuturesChains)
    {
        var continuousContractSymbol = kvp.Key;
        var chain = kvp.Value;
        foreach (var contract in chain)
        {

        }
    }
}
```

```
for slice in future_history:
    for continuous_contract_symbol, chain in slice.FuturesChains.items():
        for contract in chain:
            pass
```

To convert the `FutureHistory` object to a `DataFrame` that contains the trade and quote information of each contract, call the `GetAllData` method.

```
future_history.GetAllData()
```

To get the expiration dates of all the contracts in an `FutureHistory` object, call the `GetExpiryDates` method.

```
future_history.GetExpiryDates()
```

Plot Data

You need some [historical Futures data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(contract_symbol, datetime(2021, 12, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

   ```
   var history = qb.History<TradeBar>(contractSymbol, new DateTime(2021, 12, 1), new DateTime(2021, 12, 31), Resolution.Daily);
   ```

2. Drop the first two index levels.

   ```
   history.index = history.index.droplevel([0, 1])
   ```

3. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

4. Create a `Candlestick` .

   ```
   candlestick = go.Candlestick(x=history.index,
                                open=history['open'],
                                high=history['high'],
                                low=history['low'],
                                close=history['close'])
   ```

   ```
   var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
       history.Select(x => x.Open),
       history.Select(x => x.High),
       history.Select(x => x.Low),
       history.Select(x => x.Close),
       history.Select(x => x.EndTime)
   );
   ```

5. Create a `Layout` .

   ```
   layout = go.Layout(title=go.layout.Title(text=f'{contract_symbol.Value} OHLC'),
                      xaxis_title='Date',
                      yaxis_title='Price',
                      xaxis_rangeslider_visible=False)
   ```

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Price ($)");
   Title title = Title.init($"{contractSymbol} OHLC");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

6. Create the `Figure` .

   ```
   fig = go.Figure(data=[candlestick], layout=layout)
   ```

7. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

8. Show the `Figure` .

   ```
   fig.show()
   ```

   ```
   HTML(GenericChart.toChartHTML(chart))
   ```

   Candlestick charts display the open, high, low, and close prices of the contract.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

   ```
   history = qb.History(symbols, datetime(2021, 12, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

   ```
   var history = qb.History<OpenInterest>(contractSymbol, new DateTime(2021, 12, 1), new DateTime(2021, 12, 31));
   ```

2. Drop the first index level.

   ```
   history.index = history.index.droplevel(0)
   ```

3. Select data to plot.

   ```
   closing_prices = history['close'].unstack(level=0)
   ```

4. Rename the columns to be the `Symbol` of each contract.

   ```
   closing_prices.columns = [Symbol.GetAlias(SecurityIdentifier.Parse(x)) for x in closing_prices.columns]
   ```

5. Call the `plot` method on the `pandas` object.

   ```
   closing_prices.plot(title="Close", figsize=(15, 8))
   ```

6. Create `Line` charts.

   ```
   var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
       history.Select(x => x.EndTime),
       history.Select(x => x.Value)
   );
   ```

7. Create a `Layout` .

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Open Interest");
   Title title = Title.init($"{contractSymbol} Open Interest");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

8. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

9. Show the plot.

   ```
   plt.show()
   ```

   ```
   HTML(GenericChart.toChartHTML(chart))
   ```

   Line charts display the value of the property you selected in a time series.

3.8 Futures Options

Introduction

This page explains how to request, manipulate, and visualize historical Future Options data.

Create Subscriptions

Follow these steps to subscribe to a Futures Option contract:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Securities;
using QuantConnect.Securities.Future;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. [Subscribe to a Futures contract](#) .

```
var future = qb.AddFuture(Futures.Indices.SP500EMini, Resolution.Minute);
var startDate = new DateTime(2021,12,20);
var futuresContractSymbols = qb.FutureChainProvider.GetFutureContractList(future.Symbol, startDate);
var futuresContractSymbol = futuresContractSymbols.OrderBy(s => s.ID.Date).FirstOrDefault();
qb.AddFutureContract(futuresContractSymbol, fillForward: false);
```

```
future = qb.AddFuture(Futures.Indices.SP500EMini, Resolution.Minute)
start_date = datetime(2021,12,20)
futures_contract_symbols = qb.FutureChainProvider.GetFutureContractList(future.Symbol, start_date)
futures_contract_symbol = sorted(futures_contract_symbols, key=lambda s: s.ID.Date)[0]
qb.AddFutureContract(futures_contract_symbol, fillForward = False)
```

To view the available underlying Futures in the US Future Options dataset, see [Supported Assets](#) .

4. (Optional) Set a [contract filter](#) .

```
qb.AddFutureOption(future.Symbol, optionFilterUniverse => optionFilterUniverse.Strikes(-1, 1));
```

```
qb.AddFutureOption(future.Symbol, lambda option_filter_universe: option_filter_universe.Strikes(-1, 1))
```

The filter determines which contracts the `GetOptionHistory` method returns. If you don't set a filter, the default filter selects the contracts that have the following characteristics:

- Standard type (exclude weeklys)
- Within 1 strike price of the underlying asset price
- Expire within 31 days

If you want historical data on individual contracts and their `OpenInterest` , follow these steps to subscribe to the individual Futures Option contracts:

1. Call the `GetOptionsContractList` method with the underlying Futures Contract `Symbol` and a `datetime` `DateTime` object.

```
var fopContractSymbols = qb.OptionChainProvider.GetOptionContractList(futuresContractSymbol, startDate);
```

```
fop_contract_symbols = qb.OptionChainProvider.GetOptionContractList(futures_contract_symbol, start_date)
```

This method returns a list of `Symbol` objects that reference the Option contracts that were trading for the underlying Future contract at the given time. If you set a contract filter with `SetFilter` , it doesn't affect the results of `GetOptionContractList` .

2. Select the Symbol of the `OptionContract` object(s) for which you want to get historical data.

To filter and select contracts, you can use the following properties of each `Symbol` object:

Property	Description
`ID.Date`	The expiration date of the contract.
`ID.StrikePrice`	The strike price of the contract.
`ID.OptionRight`	The contract type. The `OptionRight` enumeration has the following members:
`ID.OptionStyle`	The contract style. The `OptionStyle` enumeration has the following members:

```
var closestExpiry = fopContractSymbols.Select(c => c.ID.Date).Min();
var fopContractSymbol = fopContractSymbols
    .Where(c => c.ID.Date == closestExpiry && c.ID.OptionRight == OptionRight.Call)
    .OrderBy(c => c.ID.StrikePrice)
    .FirstOrDefault();
```

```
closest_expiry = min([c.ID.Date for c in fop_contract_symbols])
calls = [c for c in fop_contract_symbols if c.ID.Date == closest_expiry and c.ID.OptionRight == OptionRight.Call]
fop_contract_symbol = sorted(calls, key=lambda c: c.ID.StrikePrice)[0]
```

3. Call the `AddFutureOptionContract` method with an `OptionContract` Symbol and disable fill-forward.

```
qb.AddFutureOptionContract(fopContractSymbol, fillForward: false);
```

```
qb.AddFutureOptionContract(fop_contract_symbol, fillForward = False)
```

Disable fill-forward because there are only a few `OpenInterest` data points per day.

Get Historical Data

You need a [subscription](#) before you can request historical data for Futures Option contracts. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the contract dimension, you can request historical data for a single contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Before you request historical data, call the `SetStartDate` method with a `datetime` `DateTime` to reduce the risk of [look-ahead bias](#) .

```
qb.SetStartDate(startDate);
```

```
qb.SetStartDate(start_date)
```

If you call the `SetStartDate` method, the date that you pass to the method is the latest date for which your history requests will return data.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the contract `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(fopContractSymbol, 10);
var subsetHistorySlice = qb.History(new[] {fopContractSymbol}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(fopContractSymbol, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {fopContractSymbol}, 10);
```

```
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(fopContractSymbol, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {fopContractSymbol}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(fopContractSymbol, 400);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {fopContractSymbol}, 400);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, 400);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(fop_contract_symbol, 10)
subset_history_df = qb.History([fop_contract_symbol], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, fop_contract_symbol, 10)
subset_history_trade_bar_df = qb.History(TradeBar, [fop_contract_symbol], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, fop_contract_symbol, 10)
subset_history_quote_bar_df = qb.History(QuoteBar, [fop_contract_symbol], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, fop_contract_symbol, 400)
subset_history_open_interest_df = qb.History(OpenInterest, [fop_contract_symbol], 400)
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, 400)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](fop_contract_symbol, 10)
subset_history_trade_bars = qb.History[TradeBar]([fop_contract_symbol], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](fop_contract_symbol, 10)
subset_history_quote_bars = qb.History[QuoteBar]([fop_contract_symbol], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](fop_contract_symbol, 400)
subset_history_open_interest = qb.History[OpenInterest]([fop_contract_symbol], 400)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, 400)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the contract `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(fopContractSymbol, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {fopContractSymbol}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(fopContractSymbol, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {fopContractSymbol}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(fopContractSymbol, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {fopContractSymbol}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(fopContractSymbol, TimeSpan.FromDays(2));
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {fopContractSymbol}, TimeSpan.FromDays(2));
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, TimeSpan.FromDays(2));
```

```
# DataFrame of trade and quote data
single_history_df = qb.History(fop_contract_symbol, timedelta(days=3))
subset_history_df = qb.History([fop_contract_symbol], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, fop_contract_symbol, timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, [fop_contract_symbol], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, fop_contract_symbol, timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, [fop_contract_symbol], timedelta(days=3))
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, fop_contract_symbol, timedelta(days=3))
subset_history_open_interest_df = qb.History(OpenInterest, [fop_contract_symbol], timedelta(days=3))
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, timedelta(days=3))

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](fop_contract_symbol, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([fop_contract_symbol], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](fop_contract_symbol, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([fop_contract_symbol], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](fop_contract_symbol, timedelta(days=2))
subset_history_open_interest = qb.History[OpenInterest]([fop_contract_symbol], timedelta(days=2))
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, timedelta(days=2))
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for individual Futures Option contracts during a specific period of time, call the `History` method with the Futures Option contract `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 12, 1);
var endTime = new DateTime(2021, 12, 31);

// Slice objects
var singleHistorySlice = qb.History(fopContractSymbol, startTime, endTime);
```

```
var subsetHistorySlice = qb.History(new[] {fopContractSymbol}, startTime, endTime);
var allHistorySlice = qb.History(startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(fopContractSymbol, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {fopContractSymbol}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(fopContractSymbol, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {fopContractSymbol}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>(fopContractSymbol, startTime, endTime);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {fopContractSymbol}, startTime, endTime);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, startTime, endTime);
```

```
start_time = datetime(2021, 12, 1)
end_time = datetime(2021, 12, 31)

# DataFrame of trade and quote data
single_history_df = qb.History(fop_contract_symbol, start_time, end_time)
subset_history_df = qb.History([fop_contract_symbol], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, fop_contract_symbol, start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, [fop_contract_symbol], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, fop_contract_symbol, start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, [fop_contract_symbol], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, fop_contract_symbol, start_time, end_time)
subset_history_open_interest_df = qb.History(OpenInterest, [fop_contract_symbol], start_time, end_time)
all_history_trade_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, start_time, end_time)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](fop_contract_symbol, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([fop_contract_symbol], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](fop_contract_symbol, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([fop_contract_symbol], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest](fop_contract_symbol, start_time, end_time)
subset_history_open_interest = qb.History[OpenInterest]([fop_contract_symbol], start_time, end_time)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, start_time, end_time)
```

To get historical data for all of the Futures Option contracts that traded during a specific period of time, call the `GetOptionHistory` method with the underlying Futures contract `Symbol` object, a start `DateTime` `datetime`, and an end `DateTime` `datetime`.

```
option_history = qb.GetOptionHistory(futures_contract_symbol, end_time-timedelta(days=2), end_time, Resolution.Minute, fillForward=False, extendedMarketHours=False)
```

```
var optionHistory = qb.GetOptionHistory(futuresContractSymbol, endTime-TimeSpan.FromDays(2), endTime, Resolution.Minute, fillForward: False, extendedMarketHours: False);
```

The preceding calls return data that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Future Option contract subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
Tick				
Second				
Minute				
Hour				
Daily				

Markets

The following `Market` enumeration members are available for Future Options:

Wrangle Data

You need some [historical data](#) to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some [historical data](#) to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If your history request returns a `DataFrame`, the `DataFrame` has the following index levels:

1. Contract expiry
2. Contract strike price
3. Contract type (call or put)
4. [Encoded contract Symbol](#)
5. The `EndTime` of the data sample

The columns of the `DataFrame` are the data properties. Depending on how you request data, the `DataFrame` may contain data for the underlying security, which causes some of the index levels to be an empty string for the corresponding rows.

To select the rows of the contract(s) that expire at a specific time, index the `loc` property of the `DataFrame` with the expiry time.

```
all_history_df.loc[datetime(2022, 3, 18)]
```

If you remove the first three index levels, you can index the `DataFrame` with just the contract `Symbol`, similiar to how you would with non-derivative asset classes. To remove the first three index levels, call the [droplevel](#) method.

```
all_history_df.index = all_history_df.index.droplevel([0,1,2])
```

To select the historical data of a single Futures Option contract, index the `loc` property of the `DataFrame` with the contract `Symbol`.

```
all_history_df.loc[fop_contract_symbol]
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[fop_contract_symbol]['close']
```

If you request historical data for multiple Futures Option contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the Futures Option contracts. To transform the `DataFrame` , select the column you want to display for each Futures Option contract and then call the [unstack](#)method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each security and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[fopContractSymbol].EndTime)),
    new DecimalDataFrameColumn(" Open", history.Select(x => x[fopContractSymbol].Open)),
    new DecimalDataFrameColumn(" High", history.Select(x => x[fopContractSymbol].High)),
    new DecimalDataFrameColumn(" Low", history.Select(x => x[fopContractSymbol].Low)),
    new DecimalDataFrameColumn(" Close", history.Select(x => x[fopContractSymbol].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df[" close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Futures Option subscriptions. To avoid issues, check if the `Slice` contains data for your Futures Option contract before you index it with the Futures Option `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(fopContractSymbol))
    {
        var tradeBar = slice.Bars[fopContractSymbol];
    }
    if (slice.QuoteBars.ContainsKey(fopContractSymbol))
    {
        var quoteBar = slice.QuoteBars[fopContractSymbol];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(fop_contract_symbol):
        trade_bar = slice.Bars[fop_contract_symbol]
    if slice.QuoteBars.ContainsKey(fop_contract_symbol):
        quote_bar = slice.QuoteBars[fop_contract_symbol]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol` .

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(fopContractSymbol)).Select(slice => slice.Bars[fopContractSymbol]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars` , iterate through the `TradeBars` to get the `TradeBar` of each Futures Option contract. The `TradeBars` may not have data for all of your Futures Option subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Futures Option `Symbol` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(fopContractSymbol))
    {
        var tradeBar = tradeBars[fopContractSymbol];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(fop_contract_symbol):
        trade_bar = trade_bars[fop_contract_symbol]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Futures Option contract. The `QuoteBars` may not have data for all of your Futures Option subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Futures Option `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(fopContractSymbol))
    {
        var quoteBar = quoteBars[fopContractSymbol];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(fop_contract_symbol):
        quote_bar = quote_bars[fop_contract_symbol]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

OpenInterest Objects

If the `History` method returns `OpenInterest` objects, iterate through the `OpenInterest` objects to get each one.

```
foreach (var openInterest in singleHistoryOpenInterest)
{
    Console.WriteLine(openInterest);
}
```

```
for open_interest in single_history_open_interest:
    print(open_interest)
```

If the `History` method returns a dictionary of `OpenInterest` objects, iterate through the dictionary to get the `OpenInterest` of each Futures Option contract. The dictionary of `OpenInterest` objects may not have data for all of your Futures Option contract subscriptions. To avoid issues, check if the dictionary contains data for your contract before you index it with the Futures Option contract `Symbol` .

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    if (openInterestDict.ContainsKey(fopContractSymbol))
    {
        var openInterest = openInterestDict[fopContractSymbol];
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    if open_interest_dict.ContainsKey(fop_contract_symbol):
        open_interest = open_interest_dict[fop_contract_symbol]
```

You can also iterate through each of the `OpenInterest` dictionaries.

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    foreach (var kvp in openInterestDict)
    {
        var symbol = kvp.Key;
        var openInterest = kvp.Value;
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    for kvp in open_interest_dict:
        symbol = kvp.Key
        open_interest = kvp.Value
```

OptionHistory Objects

The `GetOptionHistory` method returns an `OptionHistory` object. To get each [slice](#) in the `OptionHistory` object, iterate through it.

```
foreach (var slice in optionHistory)
{
    foreach (var kvp in slice.OptionChains)
    {
        var canonicalSymbol = kvp.Key;
        var chain = kvp.Value;
        foreach (var contract in chain)
        {

        }
    }
}
```

```
for slice in option_history:
    for canonical_symbol, chain in slice.OptionChains.items():
        for contract in chain:
            pass
```

To convert the `OptionHistory` object to a `DataFrame` that contains the trade and quote information of each contract and the underlying, call the `GetAllData` method.

```
option_history.GetAllData()
```

To get the expiration dates of all the contracts in an `OptionHistory` object, call the `GetExpiryDates` method.

```
option_history.GetExpiryDates()
```

To get the strike prices of all the contracts in an `OptionHistory` object, call the `GetStrikes` method.

```
option_history.GetStrikes()
```

Plot Data

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until we add the functionality, use Python to plot historical Future Options data.

You need to get some historical Future Options data to plot it. You can use many of the supported plotting libraries to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

```
history = qb.History(fop_contract_symbol, datetime(2021, 12, 2), datetime(2021, 12, 3))
```

```
var history = qb.History<QuoteBar>(fopContractSymbol, new DateTime(2021, 12, 2), new DateTime(2021, 12, 3));
```

2. Drop the first four index levels of the `DataFrame` that returns.

```
history.index = history.index.droplevel([0,1,2,3])
```

3. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

4. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

5. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text=f'{fop_contract_symbol.Value} OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{fopContractSymbol} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

7. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

8. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the contract.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History(fop_contract_symbols[:5], datetime(2021, 12, 2), datetime(2021, 12, 30), Resolution.Daily)
```

2. Drop the first three index levels of the returned `pandas.DataFrame` .

```
history.index = history.index.droplevel([0,1,2])
```

3. Select the data to plot.

```
closes = history['close'].unstack(level=0)
```

4. Call the `plot` method on the `pandas` object.

```
closes.plot(title="Close", figsize=(15, 5))
```

5. Create `Line` charts.

```
var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x.EndTime),
    history.Select(x => x.Close)
);
```

6. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{fopContractSymbol} Close");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

7. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

8. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.9 Forex

Introduction

This page explains how to request, manipulate, and visualize historical Forex data.

Create Subscriptions

Follow these steps to subscribe to a Forex security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddForex` method with a ticker and then save a reference to the Forex `Symbol` .

```
var eurusd = qb.AddForex("EURUSD").Symbol;
var gbpusd = qb.AddForex("GBPUSD").Symbol;
```

```
eurusd = qb.AddForex("EURUSD").Symbol
gbpusd = qb.AddForex("GBPUSD").Symbol
```

To view all of the available Forex pairs, see Supported Assets .

Get Historical Data

You need a subscription before you can request historical data for a security. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the security dimension, you can request historical data for a single Forex pair, a subset of the pairs you created subscriptions for in your notebook, or all of the pairs in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(eurusd, 10);
var subsetHistorySlice = qb.History(new[] {eurusd, gbpusd}, 10);
var allHistorySlice = qb.History(10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(eurusd, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {eurusd, gbpusd}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);
```

```
# DataFrame
single_history_df = qb.History(eurusd, 10)
subset_history_df = qb.History([eurusd, gbpusd], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](eurusd, 10)
subset_history_quote_bars = qb.History[QuoteBar]([eurusd, gbpusd], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(eurusd, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {eurusd, gbpusd}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(eurusd, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {eurusd, gbpusd}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(eurusd, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {eurusd, gbpusd}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of quote data (Forex data doesn't have trade data)
single_history_df = qb.History(eurusd, timedelta(days=3))
subset_history_df = qb.History([eurusd, gbpusd], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(eurusd, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([eurusd, gbpusd], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](eurusd, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([eurusd, gbpusd], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](eurusd, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([eurusd, gbpusd], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

The preceding calls return the most recent bars or ticks, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime datetime` , and an end `DateTime datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 2, 1);

// Slice objects
```

```
var singleHistorySlice = qb.History(eurusd, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {eurusd, gbpusd}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(eurusd, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {eurusd, gbpusd}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(eurusd, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {eurusd, gbpusd}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of quote data (Forex data doesn't have trade data)
single_history_df = qb.History(eurusd, start_time, end_time)
subset_history_df = qb.History([eurusd, gbpusd], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(eurusd, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([eurusd, gbpusd], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](eurusd, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([eurusd, gbpusd], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](eurusd, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([eurusd, gbpusd], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

The preceding calls return the bars or ticks that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Forex subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`				☐
`Second`		☐		
`Minute`		☐		
`Hour`		☐		
`Daily`		☐		

Markets

The only market available for Forex pairs is `Market.Oanda` .

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded Forex Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single Forex, index the `loc` property of the `DataFrame` with the Forex `Symbol` .

```
all_history_df.loc[eurusd]  # or all_history_df.loc['EURUSD']
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[eurusd]['close']
```

If you request historical data for multiple Forex pairs, you can transform the `DataFrame` so that it's a time series of close values for all of the Forex pairs. To transform the `DataFrame` , select the column you want to display for each Forex pair and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each Forex pair and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[eurusd].EndTime)),
    new DecimalDataFrameColumn("EURUSD Open", history.Select(x => x[eurusd].Open)),
    new DecimalDataFrameColumn("EURUSD High", history.Select(x => x[eurusd].High)),
    new DecimalDataFrameColumn("EURUSD Low", history.Select(x => x[eurusd].Low)),
    new DecimalDataFrameColumn("EURUSD Close", history.Select(x => x[eurusd].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["EURUSD close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Forex subscriptions. To avoid issues, check if the `Slice` contains data for your Forex pair before you index it with the Forex `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.QuoteBars.ContainsKey(eurusd))
    {
```

```
        var quoteBar = slice.QuoteBars[eurusd];
    }
}
```

```
for slice in all_history_slice:
        if slice.QuoteBars.ContainsKey(eurusd):
        quote_bar = slice.QuoteBars[eurusd]
```

You can also iterate through each `QuoteBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `QuoteBar` in the `Slice` for a given `Symbol` .

```
var quoteBars = allHistorySlice.Where(slice => slice.QuoteBars.ContainsKey(eurusd)).Select(slice => slice.QuoteBars[eurusd]);
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Forex pair. The `QuoteBars` may not have data for all of your Forex subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Forex `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(eurusd))
    {
        var quoteBar = quoteBars[eurusd];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(eurusd):
        quote_bar = quote_bars[eurusd]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Forex pair. The `Ticks` may not have data for all of your Forex subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Forex `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(eurusd))
    {
        var tick = ticks[eurusd];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(eurusd):
        ticks = ticks[eurusd]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some historical Forex data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

```
history = qb.History(eurusd, datetime(2021, 11, 26), datetime(2021, 12, 8), Resolution.Daily).loc[eurusd]
```

```
var history = qb.History<QuoteBar>(eurusd, new DateTime(2021, 11, 26), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='EURUSD OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{eurusd} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([eurusd, gbpusd], datetime(2021, 11, 26), datetime(2021, 12, 8), Resolution.Daily)
```

```
var history = qb.History<QuoteBar>(new [] {eurusd, gbpusd}, new DateTime(2021, 11, 26), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Select the data to plot.

```
pct_change = history['close'].unstack(0).pct_change().dropna()
```

3. Call the `plot` method on the `pandas` object.

```
pct_change.plot(title="Close Price %Change", figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[eurusd].EndTime),
    history.Select(x => x[eurusd].Close),
    Name: "EURUSD"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[gbpusd].EndTime),
    history.Select(x => x[gbpusd].Close),
    Name: "GBPUSD"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init("EURUSD & GBPUSD Close Price");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.10 CFD

Introduction

This page explains how to request, manipulate, and visualize historical CFD data.

Create Subscriptions

Follow these steps to subscribe to a CFD security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddCfd` method with a ticker and then save a reference to the CFD `Symbol` .

```
var spx = qb.AddCfd("SPX500USD").Symbol;
var usb = qb.AddCfd("USB10YUSD").Symbol;
```

```
spx = qb.AddCfd("SPX500USD").Symbol
usb = qb.AddCfd("USB10YUSD").Symbol
```

To view all of the available contracts, see Supported Assets .

Get Historical Data

You need a subscription before you can request historical data for a security. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the security dimension, you can request historical data for a single CFD contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(spx, 10);
var subsetHistorySlice = qb.History(new[] {spx, usb}, 10);
var allHistorySlice = qb.History(10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spx, 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spx, usb}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);
```

```
# DataFrame
single_history_df = qb.History(spx, 10)
subset_history_df = qb.History([spx, usb], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spx, 10)
subset_history_quote_bars = qb.History[QuoteBar]([spx, usb], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(spx, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {spx, usb}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spx, TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spx, usb}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spx, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spx, usb}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of quote data (CFD data doesn't have trade data)
single_history_df = qb.History(spx, timedelta(days=3))
subset_history_df = qb.History([spx, usb], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(spx, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([spx, usb], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spx, timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([spx, usb], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](spx, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spx, usb], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

The preceding calls return the most recent bars or ticks, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 2, 1);

// Slice objects
```

```
var singleHistorySlice = qb.History(spx, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {spx, usb}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>(spx, startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {spx, usb}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spx, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spx, usb}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of quote data (CFD data doesn't have trade data)
single_history_df = qb.History(spx, start_time, end_time)
subset_history_df = qb.History([spx, usb], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(spx, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([spx, usb], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar](spx, start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar]([spx, usb], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)

# Tick objects
single_history_ticks = qb.History[Tick](spx, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spx, usb], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

The preceding calls return the bars or ticks that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for CFD subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`				
`Second`				
`Minute`				
`Hour`				
`Daily`				

Markets

The only market available for CFD contracts is `Market.Oanda` .

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded CFD Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single CFD, index the `loc` property of the `DataFrame` with the CFD `Symbol` .

```
all_history_df.loc[spx]  # or all_history_df.loc['SPX500USD']
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[spx]['close']
```

If you request historical data for multiple CFD contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the CFD contracts. To transform the `DataFrame` , select the column you want to display for each CFD contract and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each CFD contract and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[spx].EndTime)),
    new DecimalDataFrameColumn("SPX500USD Open", history.Select(x => x[spx].Open)),
    new DecimalDataFrameColumn("SPX500USD High", history.Select(x => x[spx].High)),
    new DecimalDataFrameColumn("SPX500USD Low", history.Select(x => x[spx].Low)),
    new DecimalDataFrameColumn("SPX500USD Close", history.Select(x => x[spx].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["SPX500USD close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your CFD subscriptions. To avoid issues, check if the `Slice` contains data for your CFD contract before you index it with the CFD `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.QuoteBars.ContainsKey(spx))
    {
```

```
        var quoteBar = slice.QuoteBars[spx];
    }
}
```

```
for slice in all_history_slice:
        if slice.QuoteBars.ContainsKey(spx):
        quote_bar = slice.QuoteBars[spx]
```

You can also iterate through each `QuoteBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `QuoteBar` in the `Slice` for a given `Symbol` .

```
var quoteBars = allHistorySlice.Where(slice => slice.QuoteBars.ContainsKey(spx)).Select(slice => slice.QuoteBars[spx]);
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each CFD contract. The `QuoteBars` may not have data for all of your CFD subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the CFD `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(spx))
    {
        var quoteBar = quoteBars[spx];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(spx):
        quote_bar = quote_bars[spx]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each CFD contract. The `Ticks` may not have data for all of your CFD subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the CFD `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(spx))
    {
        var tick = ticks[spx];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(spx):
        ticks = ticks[spx]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some historical CFD data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(spx, datetime(2021, 11, 26), datetime(2021, 12, 8), Resolution.Daily).loc[spx]
   ```

```
var history = qb.History<QuoteBar>(spx, new DateTime(2021, 11, 26), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='SPX CFD OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{spx} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create the `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([spx, usb], datetime(2021, 11, 26), datetime(2021, 12, 8), Resolution.Daily)
```

```
var history = qb.History<QuoteBar>(new [] {spx, usb}, new DateTime(2021, 11, 26), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Select the data to plot.

```
pct_change = history['close'].unstack(0).pct_change().dropna()
```

3. Call the `plot` method on the `pandas` object.

```
pct_change.plot(title="Close Price %Change", figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[spx].EndTime),
    history.Select(x => x[spx].Close - x[spx].Open),
    Name: $"{spx}"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[usb].EndTime),
    history.Select(x => x[usb].Close - x[usb].Open),
    Name: $"{usb}"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{spx} & {usb} Daily Spread");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.11 Indices

Introduction

This page explains how to request, manipulate, and visualize historical Index data.

Create Subscriptions

Follow these steps to subscribe to an Index security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;
```

2. Create a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddIndex` method with a ticker and then save a reference to the Index `Symbol` .

```
var spx = qb.AddIndex("SPX").Symbol;
var vix = qb.AddIndex("VIX").Symbol;
```

```
spx = qb.AddIndex("SPX").Symbol
vix = qb.AddIndex("VIX").Symbol
```

To view all of the available indices, see [Supported Indices](#) .

Get Historical Data

You need a [subscription](#) before you can request historical data for a security. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the security dimension, you can request historical data for a single Index, a subset of the Indices you created subscriptions for in your notebook, or all of the Indices in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(spx, 10);
var subsetHistorySlice = qb.History(new[] {spx, vix}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spx, 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spx, vix}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);
```

```
# DataFrame
single_history_df = qb.History(spx, 10)
single_history_trade_bar_df = qb.History(TradeBar, spx, 10)
subset_history_df = qb.History([spx, vix], 10)
subset_history_trade_bar_df = qb.History(TradeBar, [spx, vix], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spx, 10)
subset_history_trade_bars = qb.History[TradeBar]([spx, vix], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(spx, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {spx, vix}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spx, TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spx, vix}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spx, TimeSpan.FromDays(3), Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spx, vix}, TimeSpan.FromDays(3), Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Tick);
```

```
# DataFrame of trade data (indices don't have quote data)
single_history_df = qb.History(spx, timedelta(days=3))
subset_history_df = qb.History([spx, vix], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of tick data
single_history_tick_df = qb.History(spx, timedelta(days=3), Resolution.Tick)
subset_history_tick_df = qb.History([spx, usb], timedelta(days=3), Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, timedelta(days=3), Resolution.Tick)

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spx, timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar]([spx, vix], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# Tick objects
single_history_ticks = qb.History[Tick](spx, timedelta(days=3), Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spx, vix], timedelta(days=3), Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, timedelta(days=3), Resolution.Tick)
```

The preceding calls return the most recent bars or ticks, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2021, 1, 1);
```

```
var endTime = new DateTime(2021, 2, 1);

// Slice objects
var singleHistorySlice = qb.History(spx, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {spx, vix}, startTime, endTime);
var allHistorySlice = qb.History(qb.Securities.Keys, startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>(spx, startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {spx, vix}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// Tick objects
var singleHistoryTicks = qb.History<Tick>(spx, startTime, endTime, Resolution.Tick);
var subsetHistoryTicks = qb.History<Tick>(new[] {spx, vix}, startTime, endTime, Resolution.Tick);
var allHistoryTicks = qb.History<Tick>(qb.Securities.Keys, startTime, endTime, Resolution.Tick);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 2, 1)

# DataFrame of trade data (indices don't have quote data)
single_history_df = qb.History(spx, start_time, end_time)
subset_history_df = qb.History([spx, vix], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of tick data
single_history_tick_df = qb.History(spx, start_time, end_time, Resolution.Tick)
subset_history_tick_df = qb.History([spx, vix], start_time, end_time, Resolution.Tick)
all_history_tick_df = qb.History(qb.Securities.Keys, start_time, end_time, Resolution.Tick)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar](spx, start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar]([spx, vix], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# Tick objects
single_history_ticks = qb.History[Tick](spx, start_time, end_time, Resolution.Tick)
subset_history_ticks = qb.History[Tick]([spx, vix], start_time, end_time, Resolution.Tick)
all_history_ticks = qb.History[Tick](qb.Securities.Keys, start_time, end_time, Resolution.Tick)
```

The preceding calls return the bars or ticks that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Index subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`			☐	
`Second`	☐			
`Minute`	☐			
`Hour`	☐			
`Daily`	☐			

Markets

The only market available for Indices is `Market.USA` .

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded Index Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single Index, index the `loc` property of the `DataFrame` with the Index `Symbol` .

```
all_history_df.loc[spx]  # or all_history_df.loc['SPX']
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[spx]['close']
```

If you request historical data for multiple Indices, you can transform the `DataFrame` so that it's a time series of close values for all of the Indices. To transform the `DataFrame` , select the column you want to display for each Index and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each Index and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[spx].EndTime)),
    new DecimalDataFrameColumn("SPX Open", history.Select(x => x[spx].Open)),
    new DecimalDataFrameColumn("SPX High", history.Select(x => x[spx].High)),
    new DecimalDataFrameColumn("SPX Low", history.Select(x => x[spx].Low)),
    new DecimalDataFrameColumn("SPX Close", history.Select(x => x[spx].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["SPX close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Index subscriptions. To avoid issues, check if the `Slice` contains data for your Index before you index it with the Index `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(spx))
    {
        var tradeBar = slice.Bars[spx];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(spx):
        trade_bar = slice.Bars[spx]
```

You can also iterate through each `TradeBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol` .

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(spx)).Select(slice => slice.Bars[spx]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars` , iterate through the `TradeBars` to get the `TradeBar` of each Index. The `TradeBars` may not have data for all of your Index subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Index `Symbol` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(spx))
    {
        var tradeBar = tradeBars[spx];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(spx):
        trade_bar = trade_bars[spx]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

Tick Objects

If the `History` method returns `Tick` objects, iterate through the `Tick` objects to get each one.

```
foreach (var tick in singleHistoryTicks)
{
    Console.WriteLine(tick);
}
```

```
for tick in single_history_ticks:
    print(tick)
```

If the `History` method returns `Ticks` , iterate through the `Ticks` to get the `Tick` of each Index. The `Ticks` may not have data for all of your Index subscriptions. To avoid issues, check if the `Ticks` object contains data for your security before you index it with the Index `Symbol` .

```
foreach (var ticks in allHistoryTicks)
{
    if (ticks.ContainsKey(spx))
    {
        var tick = ticks[spx];
    }
}
```

```
for ticks in all_history_ticks:
    if ticks.ContainsKey(spx):
        ticks = ticks[spx]
```

You can also iterate through each of the `Ticks` .

```
foreach (var ticks in allHistoryTicks)
{
    foreach (var kvp in ticks)
    {
        var symbol = kvp.Key;
        var tick = kvp.Value;
    }
}
```

```
for ticks in all_history_ticks:
    for kvp in ticks:
        symbol = kvp.Key
        tick = kvp.Value
```

Plot Data

You need some [historical Indices data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

```
history = qb.History(spx, datetime(2021, 11, 24), datetime(2021, 12, 8), Resolution.Daily).loc[spx]
```

```
var history = qb.History<TradeBar>(spx, new DateTime(2021, 11, 24), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Import the `plotly` `Plotly.NET` library.

```
import plotly.graph_objects as go
```

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='SPX OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{spx} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create a `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([spx, vix], datetime(2021, 11, 24), datetime(2021, 12, 8), Resolution.Daily)
```

```
var history = qb.History<TradeBar>(new [] {spx, vix}, new DateTime(2021, 11, 24), new DateTime(2021, 12, 8), Resolution.Daily);
```

2. Select the data to plot.

```
pct_change = history['close'].unstack(0).pct_change().dropna()
```

3. Call the `plot` method on the `pandas` object.

```
pct_change.plot(title="Close Price %Change", figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[spx].EndTime),
    history.Select(x => x[spx].Close - x[spx].Open),
    Name: $"{spx}"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[vix].EndTime),
    history.Select(x => x[vix].Close - x[vix].Open),
    Name: $"{vix}"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{spx} & {vix} Daily Spread");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.12 Index Options

Introduction

This page explains how to request, manipulate, and visualize historical Index Options data.

Create Subscriptions

Follow these steps to subscribe to an Index Option security:

1. Load the required assembly files and data types.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Securities;
using QuantConnect.Securities.Index;
using QuantConnect.Securities.Option;
using QuantConnect.Research;
```

2. Instantiate a `QuantBook` .

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

3. Call the `AddIndex` method with a ticker and resolution.

```
var indexSymbol = qb.AddIndex("SPX", Resolution.Minute).Symbol;
```

```
index_symbol = qb.AddIndex("SPX", Resolution.Minute).Symbol
```

To view the available indices, see [Supported Assets](#) .

If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `AddIndexOption` method with the underlying `Index Symbol` and, if you want non-standard Index Options, the [target Option ticker](#) .

```
var option = qb.AddIndexOption(indexSymbol);
```

```
option = qb.AddIndexOption(index_symbol)
```

5. *(Optional)* Set a [contract filter](#) .

```
option.SetFilter(-1, 1, 0, 90);
```

```
option.SetFilter(-1, 1, 0, 90)
```

The filter determines which contracts the `GetOptionHistory` method returns. If you don't set a filter, the default filter selects the contracts that have the following characteristics:

- Standard type (exclude weeklys)
- Within 1 strike price of the underlying asset price
- Expire within 31 days

If you want historical data on individual contracts and their `OpenInterest` , follow these steps to subscribe to individual Index Option contracts:

1. Call the `GetOptionsContractList` method with the underlying `Index Symbol` and a `datetime DateTime` .

```
var startDate = new DateTime(2021, 12, 31);

// Standard contracts
var canonicalSymbol = QuantConnect.Symbol.CreateCanonicalOption(indexSymbol, Market.USA, "?SPX");
var contractSymbols = qb.OptionChainProvider.GetOptionContractList(canonicalSymbol, startDate);

// Weekly contracts
var weeklyCanonicalSymbol = QuantConnect.Symbol.CreateCanonicalOption(indexSymbol, "SPXW", Market.USA, "?SPXW");
var weeklyContractSymbols = qb.OptionChainProvider.GetOptionContractList(weeklyCanonicalSymbol, startDate)
    .Where(s => OptionSymbol.IsWeekly(s));
```

```
start_date = datetime(2021, 12, 31)

# Standard contracts
canonical_symbol = Symbol.CreateCanonicalOption(index_symbol, Market.USA, "?SPX")
contract_symbols = qb.OptionChainProvider.GetOptionContractList(canonical_symbol, start_date)

# Weekly contracts
weekly_canonical_symbol = Symbol.CreateCanonicalOption(index_symbol, "SPXW", Market.USA, "?SPXW")
weekly_contract_symbols = qb.OptionChainProvider.GetOptionContractList(weekly_canonical_symbol, start_date)
weekly_contract_symbols = [s for s in weekly_contract_symbols if OptionSymbol.IsWeekly(s)]
```

This method returns a list of `Symbol` objects that reference the Option contracts that were trading at the given time. If you set a contract filter with `SetFilter` , it doesn't affect the results of `GetOptionContractList` .

2. Select the `Symbol` of the `OptionContract` object(s) for which you want to get historical data.

To filter and select contracts, you can use the following properties of each `Symbol` object:

Property	Description
`ID.Date`	The expiration date of the contract.
`ID.StrikePrice`	The strike price of the contract.
`ID.OptionRight`	The contract type. The `OptionRight` enumeration has the following members:
`ID.OptionStyle`	The contract style. The `OptionStyle` enumeration has the following members:

```
// Standard contracts
var contractSymbol = contractSymbols.Where(s =>
    s.ID.OptionRight == OptionRight.Call &&
    s.ID.StrikePrice == 4460 &&
    s.ID.Date == new DateTime(2022, 4, 14)).First();

// Weekly contracts
var weeklyContractSymbol = weeklyContractSymbols.Where(s =>
    s.ID.OptionRight == OptionRight.Call &&
    s.ID.StrikePrice == 4460 &&
    s.ID.Date == new DateTime(2021, 12, 31)).First();
```

```
# Standard contracts
contract_symbol = [s for s in contract_symbols
    if s.ID.OptionRight == OptionRight.Call
        and s.ID.StrikePrice == 4460
        and s.ID.Date == datetime(2022, 4, 14)][0]

# Weekly contracts
weekly_contract_symbol = [s for s in weekly_contract_symbols
    if s.ID.OptionRight == OptionRight.Call
        and s.ID.StrikePrice == 4460
        and s.ID.Date == datetime(2021, 12, 31)][0]
```

3. Call the `AddIndexOptionContract` method with an `OptionContract Symbol` and disable fill-forward.

```
qb.AddIndexOptionContract(contractSymbol, fillForward: false);
```

```
qb.AddIndexOptionContract(contract_symbol, fillForward = False)
```

Disable fill-forward because there are only a few `OpenInterest` data points per day.

Get Historical Data

You need a subscription before you can request historical data for Index Option contracts. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the contract dimension, you can request historical data for a single contract, a subset of the contracts you created subscriptions for in your notebook, or all of the contracts in your notebook.

Before you request historical data, call the `SetStartDate` method with a `datetime` `DateTime` to reduce the risk of look-ahead bias .

```
qb.SetStartDate(startDate);
```

```
qb.SetStartDate(start_date)
```

If you call the `SetStartDate` method, the date that you pass to the method is the latest date for which your history requests will return data.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the contract `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History("SPX", 10);
var subsetHistorySlice = qb.History(new[] {"SPX"}, 10);
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>("SPX", 10);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {"SPX"}, 10);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, 10);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>("SPX", 10);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {"SPX"}, 10);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, 10);

// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>("SPX", 400);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {"SPX"}, 400);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, 400);
```

```
# DataFrame of trade and quote data
single_history_df = qb.History("SPX", 10)
subset_history_df = qb.History(["SPX"], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, "SPX", 10)
subset_history_trade_bar_df = qb.History(TradeBar, ["SPX"], 10)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, 10)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, "SPX", 10)
subset_history_quote_bar_df = qb.History(QuoteBar, ["SPX"], 10)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, 10)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, "SPX", 400)
subset_history_open_interest_df = qb.History(OpenInterest, ["SPX"], 400)
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, 400)

# Slice objects
all_history_slice = qb.History(10)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar]("SPX", 10)
subset_history_trade_bars = qb.History[TradeBar](["SPX"], 10)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, 10)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar]("SPX", 10)
subset_history_quote_bars = qb.History[QuoteBar](["SPX"], 10)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, 10)

# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest]("SPX", 400)
subset_history_open_interest = qb.History[OpenInterest](["SPX"], 400)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, 400)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the contract `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History("SPX", TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {"SPX"}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>("SPX", TimeSpan.FromDays(3));
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {"SPX"}, TimeSpan.FromDays(3));
var allHistoryTradeBars = qb.History<TradeBar>(TimeSpan.FromDays(3));

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>("SPX", TimeSpan.FromDays(3), Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {"SPX"}, TimeSpan.FromDays(3), Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, TimeSpan.FromDays(3), Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>("SPX", TimeSpan.FromDays(2));
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {"SPX"}, TimeSpan.FromDays(2));
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, TimeSpan.FromDays(2));
```

```
# DataFrame of trade and quote data
single_history_df = qb.History("SPX", timedelta(days=3))
subset_history_df = qb.History(["SPX"], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, "SPX", timedelta(days=3))
subset_history_trade_bar_df = qb.History(TradeBar, ["SPX"], timedelta(days=3))
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, "SPX", timedelta(days=3))
subset_history_quote_bar_df = qb.History(QuoteBar, ["SPX"], timedelta(days=3))
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, timedelta(days=3))

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, "SPX", timedelta(days=3))
subset_history_open_interest_df = qb.History(OpenInterest, ["SPX"], timedelta(days=3))
all_history_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, timedelta(days=3))

# Slice objects
all_history_slice = qb.History(timedelta(days=3))
```

```
# TradeBar objects
single_history_trade_bars = qb.History[TradeBar]("SPX", timedelta(days=3))
subset_history_trade_bars = qb.History[TradeBar](["SPX"], timedelta(days=3))
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, timedelta(days=3))

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar]("SPX", timedelta(days=3), Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar](["SPX"], timedelta(days=3), Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, timedelta(days=3), Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest]("SPX", timedelta(days=2))
subset_history_open_interest = qb.History[OpenInterest](["SPX"], timedelta(days=2))
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, timedelta(days=2))
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for individual Index Option contracts during a specific period of time, call the `History` method with the Index Option contract `Symbol` object(s), a start `DateTime datetime` , and an end `DateTime datetime` . The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2021, 12, 1);
var endTime = new DateTime(2021, 12, 31);

// Slice objects
var singleHistorySlice = qb.History("SPX", startTime, endTime);
var subsetHistorySlice = qb.History(new[] {"SPX"}, startTime, endTime);
var allHistorySlice = qb.History(startTime, endTime);

// TradeBar objects
var singleHistoryTradeBars = qb.History<TradeBar>("SPX", startTime, endTime);
var subsetHistoryTradeBars = qb.History<TradeBar>(new[] {"SPX"}, startTime, endTime);
var allHistoryTradeBars = qb.History<TradeBar>(qb.Securities.Keys, startTime, endTime);

// QuoteBar objects
var singleHistoryQuoteBars = qb.History<QuoteBar>("SPX", startTime, endTime, Resolution.Minute);
var subsetHistoryQuoteBars = qb.History<QuoteBar>(new[] {"SPX"}, startTime, endTime, Resolution.Minute);
var allHistoryQuoteBars = qb.History<QuoteBar>(qb.Securities.Keys, startTime, endTime, Resolution.Minute);


// OpenInterest objects
var singleHistoryOpenInterest = qb.History<OpenInterest>("SPX", startTime, endTime);
var subsetHistoryOpenInterest = qb.History<OpenInterest>(new[] {"SPX"}, startTime, endTime);
var allHistoryOpenInterest = qb.History<OpenInterest>(qb.Securities.Keys, startTime, endTime);
```

```
start_time = datetime(2021, 12, 1)
end_time = datetime(2021, 12, 31)

# DataFrame of trade and quote data
single_history_df = qb.History("SPX", start_time, end_time)
subset_history_df = qb.History(["SPX"], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# DataFrame of trade data
single_history_trade_bar_df = qb.History(TradeBar, "SPX", start_time, end_time)
subset_history_trade_bar_df = qb.History(TradeBar, ["SPX"], start_time, end_time)
all_history_trade_bar_df = qb.History(TradeBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of quote data
single_history_quote_bar_df = qb.History(QuoteBar, "SPX", start_time, end_time)
subset_history_quote_bar_df = qb.History(QuoteBar, ["SPX"], start_time, end_time)
all_history_quote_bar_df = qb.History(QuoteBar, qb.Securities.Keys, start_time, end_time)

# DataFrame of open interest data
single_history_open_interest_df = qb.History(OpenInterest, "SPX", start_time, end_time)
subset_history_open_interest_df = qb.History(OpenInterest, ["SPX"], start_time, end_time)
all_history_trade_open_interest_df = qb.History(OpenInterest, qb.Securities.Keys, start_time, end_time)

# TradeBar objects
single_history_trade_bars = qb.History[TradeBar]("SPX", start_time, end_time)
subset_history_trade_bars = qb.History[TradeBar](["SPX"], start_time, end_time)
all_history_trade_bars = qb.History[TradeBar](qb.Securities.Keys, start_time, end_time)

# QuoteBar objects
single_history_quote_bars = qb.History[QuoteBar]("SPX", start_time, end_time, Resolution.Minute)
subset_history_quote_bars = qb.History[QuoteBar](["SPX"], start_time, end_time, Resolution.Minute)
all_history_quote_bars = qb.History[QuoteBar](qb.Securities.Keys, start_time, end_time, Resolution.Minute)


# OpenInterest objects
single_history_open_interest = qb.History[OpenInterest]("SPX", start_time, end_time)
subset_history_open_interest = qb.History[OpenInterest](["SPX"], start_time, end_time)
all_history_open_interest = qb.History[OpenInterest](qb.Securities.Keys, start_time, end_time)
```

To get historical data for all of the Index Option contracts that pass your [filter](#) during a specific period of time, call the `GetOptionHistory` method with the canonical Index Option `Symbol` object, a start `DateTime datetime` , and an end `DateTime datetime` .

```
option_history = qb.GetOptionHistory(option.Symbol, end_time-timedelta(days=2), end_time, Resolution.Minute, fillForward=False, extendedMarketHours=False)
```

```
var optionHistory = qb.GetOptionHistory(option.Symbol, endTime-TimeSpan.FromDays(2), endTime, Resolution.Minute, fillForward: False, extendedMarketHours: False);
```

The preceding calls return data that have a timestamp within the defined period of time.

Resolutions

The following table shows the available resolutions and data formats for Index Option contract subscriptions:

Resolution	TradeBar	QuoteBar	Trade Tick	Quote Tick
`Tick`				
`Second`				
`Minute`	☐	☐		
`Hour`	☐	☐		
`Daily`	☐	☐		

Markets

The following `Market` enumeration members are available for Index Options:

Wrangle Data

You need some [historical data](#) to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some [historical data](#) to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If your history request returns a `DataFrame` , the `DataFrame` has the following index levels:

1. Contract expiry
2. Contract strike price
3. Contract type (call or put)
4. [Encoded contract Symbol](#)
5. The `EndTime` of the data sample

The columns of the `DataFrame` are the data properties. Depending on how you request data, the `DataFrame` may contain data for the underlying security, which causes some of the index levels to be an empty string for the corresponding rows.

To select the rows of the contract(s) that expire at a specific time, index the `loc` property of the `DataFrame` with the expiry time.

```
all_history_df.loc[datetime(2022, 4, 14)]
```

If you remove the first three index levels, you can index the `DataFrame` with just the contract `Symbol` , similiar to how you would with non-derivative asset classes. To remove the first three index levels, call the [droplevel](#) method.

```
all_history_df.index = all_history_df.index.droplevel([0,1,2])
```

To select the historical data of a single Index Options contract, index the `loc` property of the `DataFrame` with the contract `Symbol` .

```
all_history_df.loc[contract_symbol]
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[contract_symbol]['close']
```

If you request historical data for multiple Index Option contracts, you can transform the `DataFrame` so that it's a time series of close values for all of the Index Option contracts. To transform the `DataFrame` , select the column you want to display for each Index Option contract and then call the [unstack](#) method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each security and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[contractSymbol].EndTime)),
    new DecimalDataFrameColumn(" Open", history.Select(x => x[contractSymbol].Open)),
    new DecimalDataFrameColumn(" High", history.Select(x => x[contractSymbol].High)),
    new DecimalDataFrameColumn(" Low", history.Select(x => x[contractSymbol].Low)),
    new DecimalDataFrameColumn(" Close", history.Select(x => x[contractSymbol].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df[" close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your Index Options subscriptions. To avoid issues, check if the `Slice` contains data for your Index Option contract before you index it with the Index Options `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.Bars.ContainsKey(contractSymbol))
    {
        var tradeBar = slice.Bars[contractSymbol];
    }
    if (slice.QuoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = slice.QuoteBars[contractSymbol];
    }
}
```

```
for slice in all_history_slice:
        if slice.Bars.ContainsKey(contract_symbol):
        trade_bar = slice.Bars[contract_symbol]
    if slice.QuoteBars.ContainsKey(contract_symbol):
        quote_bar = slice.QuoteBars[contract_symbol]
```

You can also iterate through each `TradeBar` and `QuoteBar` in the `Slice` .

```
foreach (var slice in allHistorySlice)
{
    foreach (var kvp in slice.Bars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
    foreach (var kvp in slice.QuoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for slice in all_history_slice:
    for kvp in slice.Bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
    for kvp in slice.QuoteBars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

You can also use LINQ to select each `TradeBar` in the `Slice` for a given `Symbol` .

```
var tradeBars = allHistorySlice.Where(slice => slice.Bars.ContainsKey(contractSymbol)).Select(slice => slice.Bars[contractSymbol]);
```

TradeBar Objects

If the `History` method returns `TradeBar` objects, iterate through the `TradeBar` objects to get each one.

```
foreach (var tradeBar in singleHistoryTradeBars)
{
    Console.WriteLine(tradeBar);
```

```
}
```

```
for trade_bar in single_history_trade_bars:
    print(trade_bar)
```

If the `History` method returns `TradeBars` , iterate through the `TradeBars` to get the `TradeBar` of each Index Option contract. The `TradeBars` may not have data for all of your Index Options subscriptions. To avoid issues, check if the `TradeBars` object contains data for your security before you index it with the Index Options `Symbol` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    if (tradeBars.ContainsKey(contractSymbol))
    {
        var tradeBar = tradeBars[contractSymbol];
    }
}
```

```
for trade_bars in all_history_trade_bars:
    if trade_bars.ContainsKey(contract_symbol):
        trade_bar = trade_bars[contract_symbol]
```

You can also iterate through each of the `TradeBars` .

```
foreach (var tradeBars in allHistoryTradeBars)
{
    foreach (var kvp in tradeBars)
    {
        var symbol = kvp.Key;
        var tradeBar = kvp.Value;
    }
}
```

```
for trade_bars in all_history_trade_bars:
    for kvp in trade_bars:
        symbol = kvp.Key
        trade_bar = kvp.Value
```

QuoteBar Objects

If the `History` method returns `QuoteBar` objects, iterate through the `QuoteBar` objects to get each one.

```
foreach (var quoteBar in singleHistoryQuoteBars)
{
    Console.WriteLine(quoteBar);
}
```

```
for quote_bar in single_history_quote_bars:
    print(quote_bar)
```

If the `History` method returns `QuoteBars` , iterate through the `QuoteBars` to get the `QuoteBar` of each Index Option contract. The `QuoteBars` may not have data for all of your Index Options subscriptions. To avoid issues, check if the `QuoteBars` object contains data for your security before you index it with the Index Options `Symbol` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    if (quoteBars.ContainsKey(contractSymbol))
    {
        var quoteBar = quoteBars[contractSymbol];
    }
}
```

```
for quote_bars in all_history_quote_bars:
    if quote_bars.ContainsKey(contract_symbol):
        quote_bar = quote_bars[contract_symbol]
```

You can also iterate through each of the `QuoteBars` .

```
foreach (var quoteBars in allHistoryQuoteBars)
{
    foreach (var kvp in quoteBars)
    {
        var symbol = kvp.Key;
        var quoteBar = kvp.Value;
    }
}
```

```
for quote_bars in all_history_quote_bars:
    for kvp in quote_bars:
        symbol = kvp.Key
        quote_bar = kvp.Value
```

OpenInterest Objects

If the `History` method returns `OpenInterest` objects, iterate through the `OpenInterest` objects to get each one.

```
foreach (var openInterest in singleHistoryOpenInterest)
{
    Console.WriteLine(openInterest);
}
```

```
for open_interest in single_history_open_interest:
    print(open_interest)
```

If the `History` method returns a dictionary of `OpenInterest` objects, iterate through the dictionary to get the `OpenInterest` of each Index Option contract. The dictionary of `OpenInterest` objects may not have data for all of your Index Options contract subscriptions. To avoid issues, check if the dictionary contains data for your contract before you index it with the Index Options contract `Symbol` .

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    if (openInterestDict.ContainsKey(contractSymbol))
    {
        var openInterest = openInterestDict[contractSymbol];
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    if open_interest_dict.ContainsKey(contract_symbol):
        open_interest = open_interest_dict[contract_symbol]
```

You can also iterate through each of the `OpenInterest` dictionaries.

```
foreach (var openInterestDict in allHistoryOpenInterest)
{
    foreach (var kvp in openInterestDict)
    {
        var symbol = kvp.Key;
        var openInterest = kvp.Value;
    }
}
```

```
for open_interest_dict in all_history_open_interest:
    for kvp in open_interest_dict:
        symbol = kvp.Key
        open_interest = kvp.Value
```

OptionHistory Objects

The `GetOptionHistory` method returns an `OptionHistory` object. To get each slice in the `OptionHistory` object, iterate through it.

```
foreach (var slice in optionHistory)
{
    foreach (var kvp in slice.OptionChains)
    {
        var canonicalSymbol = kvp.Key;
        var chain = kvp.Value;
        foreach (var contract in chain)
        {

        }
    }
}
```

```
for slice in option_history:
    for canonical_symbol, chain in slice.OptionChains.items():
        for contract in chain:
            pass
```

To convert the `OptionHistory` object to a `DataFrame` that contains the trade and quote information of each contract and the underlying, call the `GetAllData` method.

```
option_history.GetAllData()
```

To get the expiration dates of all the contracts in an `OptionHistory` object, call the `GetExpiryDates` method.

```
option_history.GetExpiryDates()
```

To get the strike prices of all the contracts in an `OptionHistory` object, call the `GetStrikes` method.

```
option_history.GetStrikes()
```

Plot Data

You need some [historical Index Options data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(contract_symbol, datetime(2021, 12, 30), datetime(2021, 12, 31))
   ```

   ```
   var history = qb.History<QuoteBar>(contractSymbol, new DateTime(2021, 12, 30), new DateTime(2021, 12, 30));
   ```

2. Drop the first four index levels of `DataFrame` that returns.

   ```
   history.index = history.index.droplevel([0,1,2,3])
   ```

3. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

4. Create a `Candlestick` .

   ```
   candlestick = go.Candlestick(x=history.index,
                                open=history['open'],
                                high=history['high'],
                                low=history['low'],
                                close=history['close'])
   ```

   ```
   var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
       history.Select(x => x.Open),
       history.Select(x => x.High),
       history.Select(x => x.Low),
       history.Select(x => x.Close),
       history.Select(x => x.EndTime)
   );
   ```

5. Create a `Layout` .

   ```
   layout = go.Layout(title=go.layout.Title(text=f'{symbol.Value} OHLC'),
                      xaxis_title='Date',
                      yaxis_title='Price',
                      xaxis_rangeslider_visible=False)
   ```

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Price ($)");
   Title title = Title.init($"{contractSymbol} OHLC");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

6. Create a `Figure` .

   ```
   fig = go.Figure(data=[candlestick], layout=layout)
   ```

7. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

8. Show the `Figure` .

   ```
   fig.show()
   ```

   ```
   HTML(GenericChart.toChartHTML(chart))
   ```

   The Jupyter Notebook displays a candlestick chart of the Option contract's price.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

   ```
   history = qb.History(OpenInterest, contract_symbol, datetime(2021, 12, 1), datetime(2021, 12, 31))
   ```

   ```
   var history = qb.History<OpenInterest>(contractSymbol, new DateTime(2021, 12, 1), new DateTime(2021, 12, 31));
   ```

2. Drop the first three index levels of the `DataFrame` that returns.

   ```
   history.index = history.index.droplevel([0,1,2])
   ```

3. Select the open interest data.

   ```
   history = history['openinterest'].unstack(level=0).ffill()
   ```

4. Rename the column to be the `Symbol` of each contract.

```
history.columns = [
    Symbol.GetAlias(SecurityIdentifier.Parse(x), index_symbol)
        for x in history.columns]
```

5. Call the `plot` method with a title and figure size.

```
history.plot(title="Open Interest", figsize=(16, 8))
```

6. Create `Line` charts.

```
var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x.EndTime),
    history.Select(x => x.Value)
);
```

7. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Open Interest");
Title title = Title.init($"{contractSymbol} Open Interest");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

8. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

9. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

The Jupyter Notebook displays a line chart of open interest data.

3.13 Alternative Data

Introduction

This page explains how to request, manipulate, and visualize historical alternative data. This tutorial uses the VIX Daily Price dataset from the CBOE as the example dataset.

Create Subscriptions

Follow these steps to subscribe to an alternative dataset from the Dataset Market :

1. Load the required assembly files and data types.

   ```
   #load "../Initialize.csx"
   #load "../QuantConnect.csx"

   using QuantConnect;
   using QuantConnect.Data;
   using QuantConnect.Algorithm;
   using QuantConnect.Research;
   using QuantConnect.DataSource;
   ```

2. Load the dynamic link library (DLL) of the dataset.

   To load the DLL of any dataset, type:

   ```
   #r "../QuantConnect.DataSource.<nameOfAlternativeDatasetClass>.dll"
   ```

   For example, to load the DLL of the CBOE dataset, type:

   ```
   #r "../QuantConnect.DataSource.CBOE.dll"
   ```

3. Create a `QuantBook` .

   ```
   var qb = new QuantBook();
   ```

   ```
   qb = QuantBook()
   ```

4. Call the `AddData` method with the dataset class, a ticker, and a resolution and then save a reference to the alternative data `Symbol` .

   ```
   var vix = qb.AddData<CBOE>("VIX", Resolution.Daily).Symbol;
   var v3m = qb.AddData<CBOE>("VIX3M", Resolution.Daily).Symbol;
   ```

   ```
   vix = qb.AddData(CBOE, "VIX", Resolution.Daily).Symbol
   v3m = qb.AddData(CBOE, "VIX3M", Resolution.Daily).Symbol
   ```

   To view the arguments that the `AddData` method accepts for each dataset, see the dataset listing .

   If you don't pass a resolution argument, the default resolution of the dataset is used by default. To view the supported resolutions and the default resolution of each dataset, see the dataset listing .

Get Historical Data

You need a subscription before you can request historical data for a dataset. On the time dimension, you can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time. On the dataset dimension, you can request historical data for a single dataset subscription, a subset of the dataset subscriptions you created in your notebook, or all of the dataset subscriptions in your notebook.

Trailing Number of Bars

To get historical data for a number of trailing bars, call the `History` method with the `Symbol` object(s) and an integer.

```
// Slice objects
var singleHistorySlice = qb.History(vix, 10);
var subsetHistorySlice = qb.History(new[] {vix, v3m}, 10);
var allHistorySlice = qb.History(10);

// CBOE objects
var singleHistoryDataObjects = qb.History<CBOE>(vix, 10);
var subsetHistoryDataObjects = qb.History<CBOE>(new[] {vix, v3m}, 10);
var allHistoryDataObjects = qb.History<CBOE>(qb.Securities.Keys, 10);
```

```
# DataFrame
single_history_df = qb.History(vix, 10)
subset_history_df = qb.History([vix, v3m], 10)
all_history_df = qb.History(qb.Securities.Keys, 10)

# Slice objects
all_history_slice = qb.History(10)

# CBOE objects
single_history_data_objects = qb.History[CBOE](vix, 10)
subset_history_data_objects = qb.History[CBOE]([vix, v3m], 10)
all_history_data_objects = qb.History[CBOE](qb.Securities.Keys, 10)
```

The preceding calls return the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

To get historical data for a trailing period of time, call the `History` method with the `Symbol` object(s) and a `TimeSpan` `timedelta` .

```
// Slice objects
var singleHistorySlice = qb.History(vix, TimeSpan.FromDays(3));
var subsetHistorySlice = qb.History(new[] {vix, v3m}, TimeSpan.FromDays(3));
var allHistorySlice = qb.History(10);

// CBOE objects
var singleHistoryDataObjects = qb.History<CBOE>(vix, TimeSpan.FromDays(3));
var subsetHistoryDataObjects = qb.History<CBOE>(new[] {vix, v3m}, TimeSpan.FromDays(3));
var allHistoryDataObjects = qb.History<CBOE>(TimeSpan.FromDays(3));
```

```
# DataFrame
single_history_df = qb.History(vix, timedelta(days=3))
subset_history_df = qb.History([vix, v3m], timedelta(days=3))
all_history_df = qb.History(qb.Securities.Keys, timedelta(days=3))

# Slice objects
all_history_slice = qb.History(timedelta(days=3))

# CBOE objects
single_history_data_objects = qb.History[CBOE](vix, timedelta(days=3))
subset_history_data_objects = qb.History[CBOE]([vix, v3m], timedelta(days=3))
all_history_data_objects = qb.History[CBOE](qb.Securities.Keys, timedelta(days=3))
```

The preceding calls return the most recent bars or ticks, excluding periods of time when the exchange was closed.

Defined Period of Time

To get historical data for a specific period of time, call the `History` method with the `Symbol` object(s), a start `DateTime` `datetime` , and an end `DateTime` `datetime` . The start and end times you provide are based in the notebook time zone .

```
var startTime = new DateTime(2021, 1, 1);
var endTime = new DateTime(2021, 3, 1);

// Slice objects
var singleHistorySlice = qb.History(vix, startTime, endTime);
var subsetHistorySlice = qb.History(new[] {vix, v3m}, startTime, endTime);
```

```
var allHistorySlice = qb.History(startTime, endTime);

// CBOE objects
var singleHistoryDataObjects = qb.History<CBOE>(vix, startTime, endTime);
var subsetHistoryDataObjects = qb.History<CBOE>(new[] {vix, v3m}, startTime, endTime);
var allHistoryDataObjects = qb.History<CBOE>(qb.Securities.Keys, startTime, endTime);
```

```
start_time = datetime(2021, 1, 1)
end_time = datetime(2021, 3, 1)

# DataFrame
single_history_df = qb.History(vix, start_time, end_time)
subset_history_df = qb.History([vix, v3m], start_time, end_time)
all_history_df = qb.History(qb.Securities.Keys, start_time, end_time)

# Slice objects
all_history_slice = qb.History(start_time, end_time)

# CBOE objects
single_history_data_objects = qb.History[CBOE](vix, start_time, end_time)
subset_history_data_objects = qb.History[CBOE]([vix, v3m], start_time, end_time)
all_history_data_objects = qb.History[CBOE](qb.Securities.Keys, start_time, end_time)
```

The preceding calls return the bars or ticks that have a timestamp within the defined period of time.

If you do not pass a resolution to the `History` method, the `History` method uses the resolution that the `AddData` method used when you created the subscription .

Wrangle Data

You need some historical data to perform wrangling operations. The process to manipulate the historical data depends on its data type. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some historical data to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data. The process to manipulate the historical data depends on its data type.

DataFrame Objects

If the `History` method returns a `DataFrame` , the first level of the `DataFrame` index is the encoded dataset Symbol and the second level is the `EndTime` of the data sample. The columns of the `DataFrame` are the data properties.

To select the historical data of a single dataset, index the `loc` property of the `DataFrame` with the dataset `Symbol` .

```
all_history_df.loc[vix]  # or all_history_df.loc['VIX']
```

To select a column of the `DataFrame` , index it with the column name.

```
all_history_df.loc[vix]['close']
```

If you request historical data for multiple tickers, you can transform the `DataFrame` so that it's a time series of close values for all of the tickers. To transform the `DataFrame` , select the column you want to display for each ticker and then call the unstack method.

```
all_history_df['close'].unstack(level=0)
```

The `DataFrame` is transformed so that the column indices are the `Symbol` of each ticker and each row contains the close value.

The historical data methods don't return DataFrame objects, but you can create one for efficient vectorized data wrangling.

```
using Microsoft.Data.Analysis;

var columns = new DataFrameColumn[] {
    new PrimitiveDataFrameColumn("Time", history.Select(x => x[vix].EndTime)),
    new DecimalDataFrameColumn("VIX Open", history.Select(x => x[vix].Open)),
    new DecimalDataFrameColumn("VIX High", history.Select(x => x[vix].High)),
    new DecimalDataFrameColumn("VIX Low", history.Select(x => x[vix].Low)),
    new DecimalDataFrameColumn("VIX Close", history.Select(x => x[vix].Close))
};
var df = new DataFrame(columns);
df
```

To select a particular column of the DataFrame, index it with the column name.

```
df["VIX close"]
```

Slice Objects

If the `History` method returns `Slice` objects, iterate through the `Slice` objects to get each one. The `Slice` objects may not have data for all of your dataset subscriptions. To avoid issues, check if the `Slice` contains data for your ticker before you index it with the dataset `Symbol` .

```
foreach (var slice in allHistorySlice) {
    if (slice.ContainsKey(vix))
    {
        var data = slice[vix];
    }
}
```

```
for slice in all_history_slice:
        if slice.ContainsKey(vix):
        data = slice[vix]
```

Plot Data

You need some historical alternative data to produce plots. You can use many of the supported plotting libraries `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

You can only create candlestick charts for alternative datasets that have open, high, low, and close properties.

Follow these steps to plot candlestick charts:

1. Get some historical data.

   ```
   history = qb.History(vix, datetime(2021, 1, 1), datetime(2021, 2, 1)).loc[vix]
   ```

   ```
   var history = qb.History<CBOE>(vix, new DateTime(2021, 12, 1), new DateTime(2021, 12, 31));
   ```

2. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   ```

```
using Plotly.NET.LayoutObjects;
```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text='VIX from CBOE OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{vix} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create a `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the alternative data.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Get some historical data.

```
history = qb.History([vix, v3m], datetime(2021, 1, 1), datetime(2021, 2, 1))
```

```
var history = qb.History<CBOE>(new [] {vix, v3x}, new DateTime(2021, 1, 1), new DateTime(2021, 2, 1));
```

2. Select the data to plot.

```
values = history['close'].unstack(0)
```

3. Call the `plot` method on the `pandas` object.

```
values.plot(title = 'Close', figsize=(15, 10))
```

4. Create `Line` charts.

```
var chart1 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[vix].EndTime),
    history.Select(x => x[vix].Close),
    Name: $"{vix}"
);
var chart2 = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x[v3m].EndTime),
    history.Select(x => x[v3m].Close),
    Name: $"{v3m}"
);
```

5. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{vix} & {v3m} Close Price");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

6. Combine the charts and assign the `Layout` to the chart.

```
var chart = Plotly.NET.Chart.Combine(new []{chart1, chart2});
chart.WithLayout(layout);
```

7. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

3.14 Custom Data

Introduction

This page explains how to request, manipulate, and visualize historical user-defined custom data.

Define Custom Data

You must format the data file into chronological order before you define the custom data class.

To define a custom data class, extend the `BaseData` `PythonData` class and override the [GetSource](#) and [Reader](#) methods.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Algorithm;
using QuantConnect.Research;

public class Nifty : BaseData
{
    public decimal Open;
    public decimal High;
    public decimal Low;
    public decimal Close;

    public Nifty()
    {
    }

    public override SubscriptionDataSource GetSource(SubscriptionDataConfig config, DateTime date, bool isLiveMode)
    {
        var url = "http://cdn.quantconnect.com.s3.us-east-1.amazonaws.com/uploads/CNXNIFTY.csv";
        return new SubscriptionDataSource(url, SubscriptionTransportMedium.RemoteFile);
    }

    public override BaseData Reader(SubscriptionDataConfig config, string line, DateTime date, bool isLiveMode)
    {
        var index = new Nifty();
        index.Symbol = config.Symbol;

        try
        {
            //Example File Format:
            //Date,       Open       High        Low       Close     Volume      Turnover
            //2011-09-13  7792.9    7799.9     7722.65    7748.7    116534670    6107.78
            var data = line.Split(',');
            index.Time = DateTime.Parse(data[0], CultureInfo.InvariantCulture);
            index.EndTime = index.Time.AddDays(1);
            index.Open = Convert.ToDecimal(data[1], CultureInfo.InvariantCulture);
            index.High = Convert.ToDecimal(data[2], CultureInfo.InvariantCulture);
            index.Low = Convert.ToDecimal(data[3], CultureInfo.InvariantCulture);
            index.Close = Convert.ToDecimal(data[4], CultureInfo.InvariantCulture);
            index.Value = index.Close;
        }
        catch
        {
             // Do nothing
        }
        return index;
    }
}

class Nifty(PythonData):
    '''NIFTY Custom Data Class'''
    def GetSource(self, config: SubscriptionDataConfig, date: datetime, isLiveMode: bool) -> SubscriptionDataSource:
        url = "http://cdn.quantconnect.com.s3.us-east-1.amazonaws.com/uploads/CNXNIFTY.csv"
        return SubscriptionDataSource(url, SubscriptionTransportMedium.RemoteFile)

    def Reader(self, config: SubscriptionDataConfig, line: str, date: datetime, isLiveMode: bool) -> BaseData:
        if not (line.strip() and line[0].isdigit()): return None

        # New Nifty object
        index = Nifty()
        index.Symbol = config.Symbol

        try:
            # Example File Format:
            # Date,       Open       High        Low       Close     Volume      Turnover
            # 2011-09-13  7792.9    7799.9     7722.65    7748.7    116534670    6107.78
            data = line.split(',')
            index.Time = datetime.strptime(data[0], "%Y-%m-%d")
            index.EndTime = index.Time + timedelta(days=1)
            index.Value = data[4]
            index["Open"] = float(data[1])
            index["High"] = float(data[2])
            index["Low"] = float(data[3])
            index["Close"] = float(data[4])

        except:
            pass

        return index
```

Create Subscriptions

You need to [define a custom data class](#) before you can subscribe to it.

Follow these steps to subscribe to custom dataset:

1. Create a `QuantBook` .

   ```
   var qb = new QuantBook();
   ```

   ```
   qb = QuantBook()
   ```

2. Call the `AddData` method with a ticker and then save a reference to the data `Symbol` .

   ```
   var symbol = qb.AddData<Nifty>("NIFTY").Symbol;
   ```

   ```
   symbol = qb.AddData(Nifty, "NIFTY").Symbol
   ```

   Custom data has its own resolution, so you don't need to specify it.

Get Historical Data

You need a [subscription](#) before you can request historical data for a security. You can request an amount of historical data based on a trailing number of bars, a trailing period of time, or a defined period of time.

Before you request data, call `SetStartDate` method with a `datetime` `DateTime` to reduce the risk of [look-ahead bias](#) .

```
qb.SetStartDate(2014, 7, 29);
```

```
qb.SetStartDate(2014, 7, 29)
```

If you call the `SetStartDate` method, the date that you pass to the method is the latest date for which your history requests will return data.

Trailing Number of Bars

Call the `History` method with a symbol, integer, and resolution to request historical data based on the given number of trailing bars and resolution.

```
var history = qb.History(symbol, 10);
```

```
history = qb.History(symbol, 10)
```

This method returns the most recent bars, excluding periods of time when the exchange was closed.

Trailing Period of Time

Call the `History` method with a symbol, `TimeSpan timedelta` , and resolution to request historical data based on the given trailing period of time and resolution.

```
var history = qb.History(symbol, TimeSpan.FromDays(10));
```

```
history = qb.History(symbol, timedelta(days=10))
```

This method returns the most recent bars, excluding periods of time when the exchange was closed.

Defined Period of Time

Call the `History` method with a symbol, start `DateTime datetime` , end `DateTime datetime` , and resolution to request historical data based on the defined period of time and resolution. The start and end times you provide are based in the [notebook time zone](#) .

```
var startTime = new DateTime(2013, 7, 29);
var endTime = new DateTime(2014, 7, 29);
var history = qb.History(symbol, startTime, endTime);
```

```
start_time = datetime(2013, 7, 29)
end_time = datetime(2014, 7, 29)
history = qb.History(symbol, start_time, end_time)
```

This method returns the bars that are timestamped within the defined period of time.

In all of the cases above, the `History` method returns a `DataFrame` with a `MultiIndex` .

In all of the cases above, the `History` method returns an `IEnumerable<Nifty>` for single-security requests.

Download Method

To download the data directly from the remote file location instead of using your custom data class, call the `Download` method with the data URL.

```
var content = qb.Download("http://cdn.quantconnect.com.s3.us-east-1.amazonaws.com/uploads/CNXNIFTY.csv");
```

```
content = qb.Download("http://cdn.quantconnect.com.s3.us-east-1.amazonaws.com/uploads/CNXNIFTY.csv")
```

Follow these steps to convert the content to a `DataFrame` :

1. Import the `StringIO` from the `io` library.

   ```
   from io import StringIO
   ```

2. Create a `StringIO` .

   ```
   data = StringIO(content)
   ```

3. Call the `read_csv` method.

   ```
   dataframe = pd.read_csv(data, index_col=0)
   ```

Wrangle Data

You need some [historical data](#) to perform wrangling operations. To display `pandas` objects, run a cell in a notebook with the `pandas` object as the last line. To display other data formats, call the `print` method.

You need some [historical data](#) to perform wrangling operations. Use LINQ to wrangle the data and then call the `Console.WriteLine` method in a Jupyter Notebook to display the data.

The `DataFrame` that the `History` method returns has the following index levels:

1. Dataset `Symbol`
2. The `EndTime` of the data sample

The columns of the `DataFrame` are the data properties.

To select the data of a single dataset, index the `loc` property of the `DataFrame` with the data `Symbol` .

```
history.loc[symbol]
```

To select a column of the `DataFrame` , index it with the column name.

```
history.loc[symbol]['close']
```

To get each custom data object, iterate through the result of the history request.

```
foreach(var nifty in history)
{
    Console.WriteLine($"{nifty} EndTime: {nifty.EndTime}");
}
```

Plot Data

You need some [historical custom data](#) to produce plots. You can use many of the [supported plotting libraries](#) `Plot.NET` package to visualize data in various formats. For example, you can plot candlestick and line charts.

Candlestick Chart

Follow these steps to plot candlestick charts:

1. Import the `plotly` `Plotly.NET` library.

   ```
   import plotly.graph_objects as go
   ```

   ```
   #r "../Plotly.NET.dll"
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

2. Select the data:

   ```
   history = history.loc[symbol]
   ```

   ```
   var history = qb.History<Nifty>(symbol, new DateTime(2013, 7, 1), new DateTime(2014, 7, 31));
   ```

3. Create a `Candlestick` .

```
candlestick = go.Candlestick(x=history.index,
                             open=history['open'],
                             high=history['high'],
                             low=history['low'],
                             close=history['close'])
```

```
var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
    history.Select(x => x.Open),
    history.Select(x => x.High),
    history.Select(x => x.Low),
    history.Select(x => x.Close),
    history.Select(x => x.EndTime)
);
```

4. Create a `Layout` .

```
layout = go.Layout(title=go.layout.Title(text=f'{symbol} OHLC'),
                   xaxis_title='Date',
                   yaxis_title='Price',
                   xaxis_rangeslider_visible=False)
```

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{symbol} OHLC");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Create a `Figure` .

```
fig = go.Figure(data=[candlestick], layout=layout)
```

6. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

7. Show the `Figure` .

```
fig.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Candlestick charts display the open, high, low, and close prices of the security.

Line Chart

Follow these steps to plot line charts using built-in methods `Plotly.NET` package :

1. Select data to plot.

```
values = history['value'].unstack(level=0)
```

2. Call the `plot` method on the `pandas` object.

```
values.plot(title="Value", figsize=(15, 10))
```

3. Create a `Line` chart.

```
var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
    history.Select(x => x.EndTime),
    history.Select(x => x.Close)
);
```

4. Create a `Layout` .

```
LinearAxis xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
LinearAxis yAxis = new LinearAxis();
yAxis.SetValue("title", "Price ($)");
Title title = Title.init($"{symbol} Close");

Layout layout = new Layout();
layout.SetValue("xaxis", xAxis);
layout.SetValue("yaxis", yAxis);
layout.SetValue("title", title);
```

5. Assign the `Layout` to the chart.

```
chart.WithLayout(layout);
```

6. Show the plot.

```
plt.show()
```

```
HTML(GenericChart.toChartHTML(chart))
```

Line charts display the value of the property you selected in a time series.

4 Charting

The Research Environment is centered around analyzing and understanding data. One way to gain a more intuitive understanding of the existing relationships in our data is to visualize it using charts. There are many different libraries that allow you to chart our data in different ways. Sometimes the right chart can illuminate an interesting relationship in the data. Click one of the following libraries to learn more about it:

Bokeh

Matplotlib

Plotly

Seaborn

Plotly NET

See Also

[Supported Libraries](#)
[Algorithm Charting](#)

4.1 Bokeh

Introduction

`bokeh` is a Python library you can use to create interactive visualizations. It helps you build beautiful graphics, ranging from simple plots to complex dashboards with streaming datasets. With `bokeh` , you can create JavaScript-powered visualizations without writing any JavaScript.

Import Libraries

Follow these steps to import the libraries that you need:

1. Import the `bokeh` library.

   ```
   from bokeh.plotting import figure, show
   from bokeh.models import BasicTicker, ColorBar, ColumnDataSource, LinearColorMapper
   from bokeh.palettes import Category20c
   from bokeh.transform import cumsum, transform
   from bokeh.io import output_notebook
   ```

2. Call the `output_notebook` method.

   ```
   output_notebook()
   ```

3. Import the `numpy` library.

   ```
   import numpy as np
   ```

Get Historical Data

[Get some historical market data](#) to produce the plots. For example, to get data for a bank sector ETF and some banking companies over 2021, run:

```
qb = QuantBook()
tickers = ["XLF",   # Financial Select Sector SPDR Fund
           "COF",   # Capital One Financial Corporation
           "GS",    # Goldman Sachs Group, Inc.
           "JPM",   # J P Morgan Chase & Co
           "WFC"]   # Wells Fargo & Company
symbols = [qb.AddEquity(ticker, Resolution.Daily).Symbol for ticker in tickers]
history = qb.History(symbols, datetime(2021, 1, 1), datetime(2022, 1, 1))
```

Create Candlestick Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create candlestick charts.

In this example, you create a candlestick chart that shows the open, high, low, and close prices of one of the banking securities. Follow these steps to create the candlestick chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history DataFrame` with the `symbol` .

   ```
   data = history.loc[symbol]
   ```

3. Divide the `data` into days with positive returns and days with negative returns.

   ```
   up_days = data[data['close'] > data['open']]
   down_days = data[data['open'] > data['close']]
   ```

4. Call the `figure` function with a title, axis labels and x-axis type.

   ```
   plot = figure(title=f"{symbol} OHLC", x_axis_label='Date', y_axis_label='Price', x_axis_type='datetime')
   ```

5. Call the `segment` method with the `data` timestamps, high prices, low prices, and a color.

   ```
   plot.segment(data.index, data['high'], data.index, data['low'], color="black")
   ```

   This method call plots the candlestick wicks.

6. Call the `vbar` method for the up and down days with the `data` timestamps, open prices, close prices, and a color.

   ```
   width = 12*60*60*1000
   plot.vbar(up_days.index, width, up_days['open'], up_days['close'],
             fill_color="green", line_color="green")
   plot.vbar(down_days.index, width, down_days['open'], down_days['close'],
             fill_color="red", line_color="red")
   ```

   This method call plots the candlestick bodies.

7. Call the `show` function.

   ```
   show(plot)
   ```

   The Jupyter Notebook displays the candlestick chart.

Create Line Plot

You must [import the plotting libraries](#) and [get some historical data](#) to create line charts.

In this example, you create a line chart that shows the closing price for one of the banking securities. Follow these steps to create the line chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

   ```
   close_prices = history.loc[symbol]['close']
   ```

3. Call the `figure` function with title, axis labels and x-axis type..

   ```
   plot = figure(title=f"{symbol} Close Price", x_axis_label='Date', y_axis_label='Price', x_axis_type='datetime')
   ```

4. Call the `line` method with the timestamps, `close_prices` , and some design settings.

   ```
   plot.line(close_prices.index, close_prices,
             legend_label=symbol.Value, color="blue", line_width=2)
   ```

5. Call the `show` function.

   ```
   show(plot)
   ```

   The Jupyter Notebook displays the line plot.

Create Scatter Plot

You must [import the plotting libraries](#) and [get some historical data](#) to create scatter plots.

In this example, you create a scatter plot that shows the relationship between the daily returns of two banking securities. Follow these steps to create the scatter plot:

1. Select 2 `Symbol` s.

   For example, to select the `Symbol` s of the first 2 bank stocks, run:

   ```
   symbol1 = symbols[1]
   symbol2 = symbols[2]
   ```

2. Slice the `history` DataFrame with each `Symbol` and then select the close column.

   ```
   close_price1 = history.loc[symbol1]['close']
   close_price2 = history.loc[symbol2]['close']
   ```

3. Call the `pct_change` and `dropna` methods on each `Series` .

   ```
   daily_return1 = close_price1.pct_change().dropna()
   daily_return2 = close_price2.pct_change().dropna()
   ```

4. Call the `polyfit` method with the `daily_returns1` , `daily_returns2` , and a degree.

   ```
   m, b = np.polyfit(daily_returns1, daily_returns2, deg=1)
   ```

   This method call returns the slope and intercept of the ordinary least squares regression line.

5. Call the `linspace` method with the minimum and maximum values on the x-axis.

   ```
   x = np.linspace(daily_returns1.min(), daily_returns1.max())
   ```

6. Calculate the y-axis coordinates of the regression line.

   ```
   y = m*x + b
   ```

7. Call the `figure` function with a title and axis labels.

   ```
   plot = figure(title=f"{symbol1} vs {symbol2} Daily Return",
                 x_axis_label=symbol1.Value, y_axis_label=symbol2.Value)
   ```

8. Call the `line` method with x- and y-axis values, a color, and a line width.

   ```
   plot.line(x, y, color="red", line_width=2)
   ```

   This method call plots the regression line.

9. Call the `dot` method with the `daily_returns1` , `daily_returns2` , and some design settings.

   ```
   plot.dot(daily_returns1, daily_returns2, size=20, color="navy", alpha=0.5)
   ```

   This method call plots the scatter plot dots.

10. Call the `show` function.

    ```
    show(plot)
    ```

    The Jupyter Notebook displays the scatter plot.

Create Histogram

You must import the plotting libraries and get some historical data to create histograms.

In this example, you create a histogram that shows the distribution of the daily percent returns of the bank sector ETF. In addition to the bins in the histogram, you overlay a normal distribution curve for comparison. Follow these steps to create the histogram:

1. Select the `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

   ```
   close_prices = history.loc[symbol]['close']
   ```

3. Call the `pct_change` method and then call the `dropna` method.

   ```
   daily_returns = close_prices.pct_change().dropna()
   ```

4. Call the `histogram` method with the `daily_returns` , the density argument enabled, and a number of bins.

   ```
   hist, edges = np.histogram(daily_returns, density=True, bins=20)
   ```

   This method call returns the following objects:

   - `hist` : The value of the probability density function at each bin, normalized such that the integral over the range is 1.
   - `edges` : The x-axis value of the edges of each bin.

- Call the `figure` method with a title and axis labels.

```
plot = figure(title=f"{symbol} Daily Return Distribution",
              x_axis_label='Return', y_axis_label='Frequency')
```

- Call the `quad` method with the coordinates of the bins and some design settings.

```
plot.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:],
          fill_color="navy", line_color="white", alpha=0.5)
```

This method call plots the histogram bins.

- Call the `mean` and `std` methods.

```
mean = daily_returns.mean()
std = daily_returns.std()
```

- Call the `linspace` method with the lower limit, upper limit, and number data points for the x-axis of the normal distribution curve.

```
x = np.linspace(-3*std, 3*std, 1000)
```

- Calculate the y-axis values of the normal distribution curve.

```
pdf = 1/(std * np.sqrt(2*np.pi)) * np.exp(-(x-mean)**2 / (2*std**2))
```

- Call the `line` method with the data and style of the normal distribution PDF curve.

```
plot.line(x, pdf, color="red", line_width=4,
          alpha=0.7, legend_label="Normal Distribution PDF")
```

This method call plots the normal distribution PDF curve.

- Call `show` to show the plot.

```
show(plot)
```

The Jupyter Notebook displays the histogram.

Create Bar Chart

You must import the plotting libraries and get some historical data to create bar charts.

In this example, you create a bar chart that shows the average daily percent return of the banking securities. Follow these steps to create the bar chart:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method and then multiply by 100.

   ```
   daily_returns = close_prices.pct_change() * 100
   ```

3. Call the `mean` method.

   ```
   avg_daily_returns = daily_returns.mean()
   ```

4. Call the `DataFrame` constructor with the `data Series` and then call the `reset_index` method.

   ```
   avg_daily_returns = pd.DataFrame(avg_daily_returns, columns=['avg_return']).reset_index()
   ```

5. Call the `figure` function with a title, x-axis values, and axis labels.

   ```
   plot = figure(title='Banking Stocks Average Daily % Returns', x_range=avg_daily_returns['symbol'],
                 x_axis_label='%', y_axis_label='Stocks')
   ```

6. Call the `vbar` method with the `avg_daily_returns` , x- and y-axis column names, and a bar width.

   ```
   plot.vbar(source=avg_daily_returns, x='symbol', top='avg_return', width=0.8)
   ```

7. Rotate the x-axis label and then call the `show` function.

   ```
   plot.xaxis.major_label_orientation = 0.6
   show(plot)
   ```

   The Jupyter Notebook displays the bar chart.

Create Heat Map

You must import the plotting libraries and get some historical data to create heat maps.

In this example, you create a heat map that shows the correlation between the daily returns of the banking securities. Follow these steps to create the heat map:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method.

   ```
   daily_returns = close_prices.pct_change()
   ```

3. Call the `corr` method.

   ```
   corr_matrix = daily_returns.corr()
   ```

4. Set the index and columns of the `corr_matrix` to the ticker of each security and then set the name of the column and row indices.

   ```
   corr_matrix.index = corr_matrix.columns = [symbol.Value for symbol in symbols]
   corr_matrix.index.name = 'symbol'
   corr_matrix.columns.name = "stocks"
   ```

5. Call the `stack` , `rename` , and `reset_index` methods.

   ```
   corr_matrix = corr_matrix.stack().rename("value").reset_index()
   ```

6. Call the `figure` function with a title, axis ticks, and some design settings.

   ```
   plot = figure(title=f"Banking Stocks and Bank Sector ETF Correlation Heat Map",
                 x_range=list(corr_matrix.symbol.drop_duplicates()),
                 y_range=list(corr_matrix.stocks.drop_duplicates()),
                 toolbar_location=None,
                 tools="",
                 x_axis_location="above")
   ```

7. Select a color palette and then call the `LinearColorMapper` constructor with the color pallet, the minimum correlation, and the maximum correlation.

   ```
   colors = Category20c[len(corr_matrix.columns)]
   mapper = LinearColorMapper(palette=colors, low=corr_matrix.value.min(),
                              high=corr_matrix.value.max())
   ```

8. Call the `rect` method with the correlation plot data and design setting.

   ```
   plot.rect(source=ColumnDataSource(corr_matrix),
             x="stocks",
             y="symbol",
             width=1,
             height=1,
             line_color=None,
             fill_color=transform('value', mapper))
   ```

9. Call the `ColorBar` constructor with the `mapper` , a location, and a `BaseTicker` .

   ```
   color_bar = ColorBar(color_mapper=mapper,
                        location=(0, 0),
                        ticker=BasicTicker(desired_num_ticks=len(colors)))
   ```

   This snippet creates a color bar to represent the correlation coefficients of the heat map cells.

10. Call the `add_layout` method with the `color_bar` and a location.

    ```
    plot.add_layout(color_bar, 'right')
    ```

    This method call plots the color bar to the right of the heat map.

11. Call the `show` function.

    ```
    show(plot)
    ```

    The Jupyter Notebook displays the heat map.

Create Pie Chart

You must import the plotting libraries and get some historical data to create pie charts.

In this example, you create a pie chart that shows the weights of the banking securities in a portfolio if you allocate to them based on their inverse volatility. Follow these steps to create the pie chart:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method.

   ```
   daily_returns = close_prices.pct_change()
   ```

3. Call the `var` method, take the inverse, and then normalize the result.

```
inverse_variance = 1 / daily_returns.var()
inverse_variance /= np.sum(inverse_variance)   # Normalization
inverse_variance *= np.pi*2    # For a full circle circumference in radian
```

4. Call the `DataFrame` constructor with the `inverse_variance Series` and then call the `reset_index` method.

   ```
   inverse_variance = pd.DataFrame(inverse_variance, columns=["inverse variance"]).reset_index()
   ```

5. Add a color column to the `inverse_variance DataFrame` .

   ```
   inverse_variance['color'] = Category20c[len(inverse_variance.index)]
   ```

6. Call the `figure` function with a title.

   ```
   plot = figure(title=f"Banking Stocks and Bank Sector ETF Allocation")
   ```

7. Call the `wedge` method with design settings and the `inverse_variance DataFrame` .

   ```
   plot.wedge(x=0, y=1, radius=0.6, start_angle=cumsum('inverse variance', include_zero=True),
              end_angle=cumsum('inverse variance'), line_color="white", fill_color='color',
              legend_field='symbol', source=inverse_variance)
   ```

8. Call the `show` function.

   ```
   show(plot)
   ```

   The Jupyter Notebook displays the pie chart.

4.2 Matplotlib

Introduction

`matplotlib` is the most popular 2d-charting library for python. It allows you to easily create histograms, scatter plots, and various other charts. In addition, `pandas` is integrated with `matplotlib` , so you can seamlessly move between data manipulation and data visualization. This makes `matplotlib` great for quickly producing a chart to visualize your data.

Import Libraries

Follow these steps to import the libraries that you need:

1. Import the `matplotlib` , `mplfinance` , and `numpy` libraries.

   ```
   import matplotlib.pyplot as plt
   import mplfinance
   import numpy as np
   ```

2. Import, and then call, the `register_matplotlib_converters` method.

   ```
   from pandas.plotting import register_matplotlib_converters
   register_matplotlib_converters()
   ```

Get Historical Data

Get some historical market data to produce the plots. For example, to get data for a bank sector ETF and some banking companies over 2021, run:

```
qb = QuantBook()
tickers = ["XLF",   # Financial Select Sector SPDR Fund
           "COF",   # Capital One Financial Corporation
           "GS",    # Goldman Sachs Group, Inc.
           "JPM",   # J P Morgan Chase & Co
           "WFC"]   # Wells Fargo & Company
symbols = [qb.AddEquity(ticker, Resolution.Daily).Symbol for ticker in tickers]
history = qb.History(symbols, datetime(2021, 1, 1), datetime(2022, 1, 1))
```

Create Candlestick Chart

You must import the plotting libraries and get some historical data to create candlestick charts.

In this example, we'll create a candlestick chart that shows the open, high, low, and close prices of one of the banking securities. Follow these steps to create the candlestick chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history DataFrame` with the `symbol` .

   ```
   data = history.loc[symbol]
   ```

3. Rename the columns.

   ```
   data.columns = ['Close', 'High', 'Low', 'Open', 'Volume']
   ```

4. Call the `plot` method with the `data` , chart type, style, title, y-axis label, and figure size.

   ```
   mplfinance.plot(data,
                   type='candle',
                   style='charles',
                   title=f'{symbol.Value} OHLC',
                   ylabel='Price ($)',
                   figratio=(15, 10))
   ```

   The Jupyter Notebook displays the candlestick chart.

Create Line Plot

You must import the plotting libraries and get some historical data to create line charts.

In this example, you create a line chart that shows the closing price for one of the banking securities. Follow these steps to create the line chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history` DataFrame with `symbol` and then select the close column.

   ```
   data = history.loc[symbol]['close']
   ```

3. Call the `plot` method with a title and figure size.

   ```
   data.plot(title=f"{symbol} Close Price", figsize=(15, 10));
   ```

   The Jupyter Notebook displays the line plot.

Create Scatter Plot

You must import the plotting libraries and get some historical data to create scatter plots.

In this example, you create a scatter plot that shows the relationship between the daily returns of two banking securities. Follow these steps to create the scatter plot:

1. Select the 2 `Symbol` s.

   For example, to select the `Symbol` s of the first 2 bank stocks, run:

   ```
   symbol1 = symbols[1]
   symbol2 = symbols[2]
   ```

2. Slice the `history` DataFrame with each `Symbol` and then select the close column.

   ```
   close_price1 = history.loc[symbol1]['close']
   close_price2 = history.loc[symbol2]['close']
   ```

3. Call the `pct_change` and `dropna` methods on each `Series` .

   ```
   daily_returns1 = close_price1.pct_change().dropna()
   daily_returns2 = close_price2.pct_change().dropna()
   ```

4. Call the `polyfit` method with the `daily_returns1` , `daily_returns2` , and a degree.

   ```
   m, b = np.polyfit(daily_returns1, daily_returns2, deg=1)
   ```

   This method call returns the slope and intercept of the ordinary least squares regression line.

5. Call the `linspace` method with the minimum and maximum values on the x-axis.

   ```
   x = np.linspace(daily_returns1.min(), daily_returns1.max())
   ```

6. Calculate the y-axis coordinates of the regression line.

```
y = m*x + b
```

7. Call the `plot` method with the coordinates and color of the regression line.

   ```
   plt.plot(x, y, color='red')
   ```

8. In the same cell that you called the `plot` method, call the `scatter` method with the 2 daily return series.

   ```
   plt.scatter(daily_returns1, daily_returns2)
   ```

9. In the same cell that you called the `scatter` method, call the `title` , `xlabel` , and `ylabel` methods with a title and axis labels.

   ```
   plt.title(f'{symbol1} vs {symbol2} daily returns Scatter Plot')
   plt.xlabel(symbol1.Value)
   plt.ylabel(symbol2.Value);
   ```

   The Jupyter Notebook displays the scatter plot.

Create Histogram

You must import the plotting libraries and get some historical data to create histograms.

In this example, you create a histogram that shows the distribution of the daily percent returns of the bank sector ETF. In addition to the bins in the histogram, you overlay a normal distribution curve for comparison. Follow these steps to create the histogram:

1. Select the `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

   ```
   close_prices = history.loc[symbol]['close']
   ```

3. Call the `pct_change` method and then call the `dropna` method.

   ```
   daily_returns = close_prices.pct_change().dropna()
   ```

4. Call the `mean` and `std` methods.

   ```
   mean = daily_returns.mean()
   std = daily_returns.std()
   ```

5. Call the `linspace` method with the lower limit, upper limit, and number data points for the x-axis of the normal distribution curve.

   ```
   x = np.linspace(-3*std, 3*std, 1000)
   ```

6. Calculate the y-axis values of the normal distribution curve.

   ```
   pdf = 1/(std * np.sqrt(2*np.pi)) * np.exp(-(x-mean)**2 / (2*std**2))
   ```

7. Call the `plot` method with the data for the normal distribution curve.

   ```
   plt.plot(x, pdf, label="Normal Distribution")
   ```

8. In the same cell that you called the `plot` method, call the `hist` method with the daily return data and the number of bins.

   ```
   plt.hist(daily_returns, bins=20)
   ```

9. In the same cell that you called the `hist` method, call the `title` , `xlabel` , and `ylabel` methods with a title and the axis labels.

   ```
   plt.title(f'{symbol} Return Distribution')
   plt.xlabel('Daily Return')
   plt.ylabel('Count');
   ```

   The Jupyter Notebook displays the histogram.

Create Bar Chart

You must import the plotting libraries and get some historical data to create bar charts.

In this example, you create a bar chart that shows the average daily percent return of the banking securities. Follow these steps to create the bar chart:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method and then multiply by 100.

   ```
   daily_returns = close_prices.pct_change() * 100
   ```

3. Call the `mean` method.

   ```
   avg_daily_returns = daily_returns.mean()
   ```

4. Call the `figure` method with a figure size.

   ```
   plt.figure(figsize=(15, 10))
   ```

5. Call the `bar` method with the x-axis and y-axis values.

   ```
   plt.bar(avg_daily_returns.index, avg_daily_returns)
   ```

6. In the same cell that you called the `bar` method, call the `title` , `xlabel` , and `ylabel` methods with a title and the axis labels.

   ```
   plt.title('Banking Stocks Average Daily % Returns')
   plt.xlabel('Tickers')
   plt.ylabel('%');
   ```

   The Jupyter Notebook displays the bar chart.

Create Heat Map

You must import the plotting libraries and get some historical data to create heat maps.

In this example, you create a heat map that shows the correlation between the daily returns of the banking securities. Follow these steps to create the heat map:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method.

   ```
   daily_returns = close_prices.pct_change()
   ```

3. Call the `corr` method.

   ```
   corr_matrix = daily_returns.corr()
   ```

4. Call the `imshow` method with the correlation matrix, a color map, and an interpolation method.

   ```
   plt.imshow(corr_matrix, cmap='hot', interpolation='nearest')
   ```

5. In the same cell that you called the `imshow` method, call the `title`, `xticks`, and `yticks`, methods with a title and the axis tick labels.

   ```
   plt.title('Banking Stocks and Bank Sector ETF Correlation Heat Map')
   plt.xticks(np.arange(len(tickers)), labels=tickers)
   plt.yticks(np.arange(len(tickers)), labels=tickers)
   ```

6. In the same cell that you called the `imshow` method, call the `colorbar` method.

   ```
   plt.colorbar();
   ```

   The Jupyter Notebook displays the heat map.

Create Pie Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create pie charts.

In this example, you create a pie chart that shows the weights of the banking securities in a portfolio if you allocate to them based on their inverse volatility. Follow these steps to create the pie chart:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method.

   ```
   daily_returns = close_prices.pct_change()
   ```

3. Call the `var` method and then take the inverse.

   ```
   inverse_variance = 1 / daily_returns.var()
   ```

4. Call the `pie` method with the `inverse_variance Series`, the plot labels, and a display format.

   ```
   plt.pie(inverse_variance, labels=inverse_variance.index, autopct='%1.1f%%')
   ```

5. In the cell that you called the `pie` method, call the `title` method with a title.

   ```
   plt.title('Banking Stocks and Bank Sector ETF Allocation');
   ```

   The Jupyter Notebook displays the pie chart.

4.3 Plotly

Introduction

`plotly` is an online charting tool with a python API. It offers the ability to create rich and interactive graphs.

Import Libraries

Import the `plotly` library.

```
import plotly.express as px
import plotly.graph_objects as go
```

Get Historical Data

[Get some historical market data](#) to produce the plots. For example, to get data for a bank sector ETF and some banking companies over 2021, run:

```
qb = QuantBook()
tickers = ["XLF",   # Financial Select Sector SPDR Fund
           "COF",   # Capital One Financial Corporation
           "GS",    # Goldman Sachs Group, Inc.
           "JPM",   # J P Morgan Chase & Co
           "WFC"]   # Wells Fargo & Company
symbols = [qb.AddEquity(ticker, Resolution.Daily).Symbol for ticker in tickers]
history = qb.History(symbols, datetime(2021, 1, 1), datetime(2022, 1, 1))
```

Create Candlestick Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create candlestick charts.

In this example, you create a candlestick chart that shows the open, high, low, and close prices of one of the banking securities. Follow these steps to create the candlestick chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history DataFrame` with the `symbol` .

   ```
   data = history.loc[symbol]
   ```

3. Call the `Candlestick` constructor with the time and open, high, low, and close price `Series` .

   ```
   candlestick = go.Candlestick(x=data.index,
                                open=data['open'],
                                high=data['high'],
                                low=data['low'],
                                close=data['close'])
   ```

4. Call the `Layout` constructor with a title and axes labels.

   ```
   layout = go.Layout(title=go.layout.Title(text=f'{symbol.Value} OHLC'),
                      xaxis_title='Date',
                      yaxis_title='Price',
                      xaxis_rangeslider_visible=False)
   ```

5. Call the `Figure` constructor with the `candlestick` and `layout` .

   ```
   fig = go.Figure(data=[candlestick], layout=layout)
   ```

6. Call the `show` method.

   ```
   fig.show()
   ```

   The Jupyter Notebook displays the candlestick chart.

Create Line Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create line charts.

In this example, you create a line chart that shows the closing price for one of the banking securities. Follow these steps to create the line chart:

1. Select a `Symbol` .

   ```
   symbol = symbols[0]
   ```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

   ```
   data = history.loc[symbol]['close']
   ```

3. Call the `DataFrame` constructor with the `data Series` and then call the `reset_index` method.

   ```
   data = pd.DataFrame(data).reset_index()
   ```

4. Call the `line` method with `data` , the column names of the x- and y-axis in `data` , and the plot title.

   ```
   fig = px.line(data, x='time', y='close', title=f'{symbol} Close price')
   ```

5. Call the `show` method.

   ```
   fig.show()
   ```

   The Jupyter Notebook displays the line chart.

Create Scatter Plot

You must [import the plotting libraries](#) and [get some historical data](#) to create scatter plots.

In this example, you create a scatter plot that shows the relationship between the daily returns of two banking securities. Follow these steps to create the scatter plot:

1. Select 2 `Symbol` s.

   For example, to select the `Symbol` s of the first 2 bank stocks, run:

   ```
   symbol1 = symbols[1]
   symbol2 = symbols[2]
   ```

2. Slice the `history` DataFrame with each `Symbol` and then select the close column.

   ```
   close_price1 = history.loc[symbol1]['close']
   close_price2 = history.loc[symbol2]['close']
   ```

3. Call the `pct_change` and `dropna` methods on each `Series` .

   ```
   daily_return1 = close_price1.pct_change().dropna()
   daily_return2 = close_price2.pct_change().dropna()
   ```

4. Call the `scatter` method with the 2 return `Series` , the trendline option, and axes labels.

   ```
   fig = px.scatter(x=daily_return1, y=daily_return2, trendline='ols',
                    labels={'x': symbol1.Value, 'y': symbol2.Value})
   ```

5. Call the `update_layout` method with a title.

```
fig.update_layout(title=f'{symbol1.Value} vs {symbol2.Value} Daily % Returns');
```

6. Call the `show` method.

```
fig.show()
```

The Jupyter Notebook displays the scatter plot.

Create Histogram

You must import the plotting libraries and get some historical data to create histograms.

In this example, you create a histogram that shows the distribution of the daily percent returns of the bank sector ETF. Follow these steps to create the histogram:

1. Select the `Symbol` .

```
symbol = symbols[0]
```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

```
data = history.loc[symbol]['close']
```

3. Call the `pct_change` method and then call the `dropna` method.

```
daily_returns = data.pct_change().dropna()
```

4. Call the `DataFrame` constructor with the `data Series` and then call the `reset_index` method.

```
daily_returns = pd.DataFrame(daily_returns).reset_index()
```

5. Call the `histogram` method with the `daily_returns` DataFrame, the x-axis label, a title, and the number of bins.

```
fig = px.histogram(daily_returns, x='close',
                   title=f'{symbol} Daily Return of Close Price Distribution',
                   nbins=20)
```

6. Call the `show` method.

```
fig.show()
```

The Jupyter Notebook displays the histogram.

Create Bar Chart

You must import the plotting libraries and get some historical data to create bar charts.

In this example, you create a bar chart that shows the average daily percent return of the banking securities. Follow these steps to create the bar chart:

1. Select the close column and then call the `unstack` method.

```
close_prices = history['close'].unstack(level=0)
```

2. Call the `pct_change` method and then multiply by 100.

```
daily_returns = close_prices.pct_change() * 100
```

3. Call the `mean` method.

```
avg_daily_returns = daily_returns.mean()
```

4. Call the `DataFrame` constructor with the `avg_daily_returns Series` and then call the `reset_index` method.

```
avg_daily_returns = pd.DataFrame(avg_daily_returns, columns=["avg_daily_ret"]).reset_index()
```

5. Call the `bar` method with the `avg_daily_returns` and the axes column names.

```
fig = px.bar(avg_daily_returns, x='symbol', y='avg_daily_ret')
```

6. Call the `update_layout` method with a title.

```
fig.update_layout(title='Banking Stocks Average Daily % Returns');
```

7. Call the `show` method.

```
fig.show()
```

The Jupyter Notebook displays the bar plot.

Create Heat Map

You must import the plotting libraries and get some historical data to create heat maps.

In this example, you create a heat map that shows the correlation between the daily returns of the banking securities. Follow these steps to create the heat map:

1. Select the close column and then call the `unstack` method.

```
close_prices = history['close'].unstack(level=0)
```

2. Call the `pct_change` method.

```
daily_returns = close_prices.pct_change()
```

3. Call the `corr` method.

```
corr_matrix = daily_returns.corr()
```

4. Call the `imshow` method with the `corr_matrix` and the axes labels.

```
fig = px.imshow(corr_matrix, x=tickers, y=tickers)
```

5. Call the `update_layout` method with a title.

```
fig.update_layout(title='Banking Stocks and bank sector ETF Correlation Heat Map');
```

6. Call the `show` method.

```
fig.show()
```

The Jupyter Notebook displays the heat map.

Create Pie Chart

You must import the plotting libraries and get some historical data to create pie charts.

In this example, you create a pie chart that shows the weights of the banking securities in a portfolio if you allocate to them based on their inverse volatility. Follow these steps to create the pie chart:

1. Select the close column and then call the `unstack` method.

   ```
   close_prices = history['close'].unstack(level=0)
   ```

2. Call the `pct_change` method.

   ```
   daily_returns = close_prices.pct_change()
   ```

3. Call the `var` method and then take the inverse.

   ```
   inverse_variance = 1 / daily_returns.var()
   ```

4. Call the `DataFrame` constructor with the `inverse_variance Series` and then call the `reset_index` method.

   ```
   inverse_variance = pd.DataFrame(inverse_variance, columns=["inverse variance"]).reset_index()
   ```

5. Call the `pie` method with the `inverse_variance DataFrame`, the column name of the values, and the column name of the names.

   ```
   fig = px.pie(inverse_variance, values='inverse variance', names='symbol')
   ```

6. Call the `update_layout` method with a title.

   ```
   fig.update_layout(title='Asset Allocation of bank stocks and bank sector ETF');
   ```

7. Call the `show` method.

   ```
   fig.show()
   ```

   The Jupyter Notebook displays the pie chart.

4.4 Seaborn

Introduction

`seaborn` is a data visualization library based on `matplotlib` . It makes it easier to create more complicated plots and allows us to create much more visually-appealing charts than `matplotlib` charts.

Import Libraries

Follow these steps to import the libraries that you need:

1. Import the `seaborn` and `matplotlib` libraries.

```
import seaborn as sns
import matplotlib.pyplot as plt
```

2. Import, and then call, the `register_matplotlib_converters` method.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Get Historical Data

[Get some historical market data](#) to produce the plots. For example, to get data for a bank sector ETF and some banking companies over 2021, run:

```
qb = QuantBook()
tickers = ["XLF",   # Financial Select Sector SPDR Fund
           "COF",   # Capital One Financial Corporation
           "GS",    # Goldman Sachs Group, Inc.
           "JPM",   # J P Morgan Chase & Co
           "WFC"]   # Wells Fargo & Company
symbols = [qb.AddEquity(ticker, Resolution.Daily).Symbol for ticker in tickers]
history = qb.History(symbols, datetime(2021, 1, 1), datetime(2022, 1, 1))
```

Create Candlestick Chart

Seaborn does not currently support candlestick charts. Use one of the other plotting libraries to create candlestick charts.

Create Line Chart

You must import the plotting libraries and get some historical data to create line charts.

In this example, you create a line chart that shows the closing price for one of the banking securities. Follow these steps to create the chart:

1. Select a `Symbol` .

```
symbol = symbols[0]
```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

```
data = history.loc[symbol]['close']
```

3. Call the `DataFrame` constructor with the `data Series` and then call the `reset_index` method.

```
data = pd.DataFrame(data).reset_index()
```

4. Call the `lineplot` method with the `data Series` and the column name of each axis.

```
plot = sns.lineplot(data=data,
                    x='time',
                    y='close')
```

5. In the same cell that you called the `lineplot` method, call the `set` method with the y-axis label and a title.

```
plot.set(ylabel="price", title=f"{symbol} Price Over Time");
```

The Jupyter Notebook displays the line chart.

Create Scatter Plot

You must import the plotting libraries and get some historical data to create scatter plots.

In this example, you create a scatter plot that shows the relationship between the daily returns of two banking securities. Follow these steps to create the scatter plot:

1. Select 2 `Symbol` s.

For example, to select the `Symbol` s of the first 2 bank stocks, run:

```
symbol1 = symbols[1]
symbol2 = symbols[2]
```

2. Select the close column of the `history` DataFrame, call the `unstack` method, and then select the `symbol1` and `symbol2` columns.

```
close_prices = history['close'].unstack(0)[[symbol1, symbol2]]
```

3. Call the `pct_change` method and then call the `dropna` method.

```
daily_returns = close_prices.pct_change().dropna()
```

4. Call the `regplot` method with the `daily_returns DataFrame` and the column names.

```
plot = sns.regplot(data=daily_returns,
                   x=daily_returns.columns[0],
                   y=daily_returns.columns[1])
```

5. In the same cell that you called the `regplot` method, call the `set` method with the axis labels and a title.

```
plot.set(xlabel=f'{daily_returns.columns[0]} % Returns',
         ylabel=f'{daily_returns.columns[1]} % Returns',
         title=f'{symbol1} vs {symbol2} Daily % Returns');
```

The Jupyter Notebook displays the scatter plot.

Create Histogram

You must import the plotting libraries and get some historical data to create histograms.

In this example, you create a histogram that shows the distribution of the daily percent returns of the bank sector ETF. Follow these steps to create the histogram:

1. Select the `Symbol` .

```
symbol = symbols[0]
```

2. Slice the `history` DataFrame with the `symbol` and then select the close column.

```
data = history.loc[symbol]['close']
```

3. Call the `pct_change` method and then call the `dropna` method.

```
daily_returns = data.pct_change().dropna()
```

4. Call the `DataFrame` constructor with the `daily_returns Series` and then call the `reset_index` method.

```
daily_returns = pd.DataFrame(daily_returns).reset_index()
```

5. Call the `histplot` method with the `daily_returns` , the close column name, and the number of bins.

```
plot = sns.histplot(daily_returns, x='close', bins=20)
```

6. In the same cell that you called the `histplot` method, call the `set` method with the axis labels and a title.

```
plot.set(xlabel='Return',
         ylabel='Frequency',
         title=f'{symbol} Daily Return of Close Price Distribution');
```

The Jupyter Notebook displays the histogram.

Create Bar Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create bar charts.

In this example, you create a bar chart that shows the average daily percent return of the banking securities. Follow these steps to create the bar chart:

1. Select the close column and then call the `unstack` method.

```
close_prices = history['close'].unstack(level=0)
```

2. Call the `pct_change` method and then multiply by 100.

```
daily_returns = close_prices.pct_change() * 100
```

3. Call the `mean` method.

```
avg_daily_returns = daily_returns.mean()
```

4. Call the `DataFrame` constructor with the `avg_daily_returns Series` and then call the `reset_index` method.

```
avg_daily_returns = pd.DataFrame(avg_daily_returns, columns=["avg_daily_ret"]).reset_index()
```

5. Call `barplot` method with the `avg_daily_returns Series` and the axes column names.

```
plot = sns.barplot(data=avg_daily_returns, x='symbol', y='avg_daily_ret')
```

6. In the same cell that you called the `barplot` method, call the `set` method with the axis labels and a title.

```
plot.set(xlabel='Tickers',
         ylabel='%',
         title='Banking Stocks Average Daily % Returns')
```

7. In the same cell that you called the `set` method, call the `tick_params` method to rotate the x-axis labels.

```
plot.tick_params(axis='x', rotation=90)
```

The Jupyter Notebook displays the bar chart.

Create Heat Map

You must [import the plotting libraries](#) and [get some historical data](#) to create heat maps.

In this example, you create a heat map that shows the correlation between the daily returns of the banking securities. Follow these steps to create the heat map:

1. Select the close column and then call the `unstack` method.

```
close_prices = history['close'].unstack(level=0)
```

2. Call the `pct_change` method.

```
daily_returns = close_prices.pct_change()
```

3. Call the `corr` method.

```
corr_matrix = daily_returns.corr()
```

4. Call the `heatmap` method with the `corr_matrix` and the annotation argument enabled.

```
plot = sns.heatmap(corr_matrix, annot=True)
```

5. In the same cell that you called the `heatmap` method, call the `set` method with a title.

```
plot.set(title='Bank Stocks and Bank Sector ETF Correlation Coefficients');
```

The Jupyter Notebook displays the heat map.

Create Pie Chart

You must [import the plotting libraries](#) and [get some historical data](#) to create pie charts.

In this example, you create a pie chart that shows the weights of the banking securities in a portfolio if you allocate to them based on their inverse volatility. Follow these steps to create the pie chart:

1. Select the close column and then call the `unstack` method.

```
close_prices = history['close'].unstack(level=0)
```

2. Call the `pct_change` method.

```
daily_returns = close_prices.pct_change()
```

3. Call `var` method and then take the inverse.

```
inverse_variance = 1 / daily_returns.var()
```

4. Call the `color_palette` method with a palette name and then truncate the returned colors to so that you have one color for each security.

```
colors = sns.color_palette('pastel')[:len(inverse_variance.index)]
```

5. Call the `pie` method with the security weights, labels, and colors.

```
plt.pie(inverse_variance, labels=inverse_variance.index, colors=colors, autopct='%1.1f%%')
```

6. In the same cell that you called the `pie` method, call the `title` method with a title.

```
plt.title(title='Banking Stocks and Bank Sector ETF Allocation');
```

The Jupyter Notebook displays the pie chart.

4.5 Plotly NET

Introduction

`Plotly.NET` provides functions for generating and rendering plotly.js charts in `.NET` programming languages. Our .NET interactive notebooks support its C# implementation.

Import Libraries

Follow these steps to import the libraries that you need:

1. Load the necessary assembly files.

   ```
   #r "../Plotly.NET.dll"
   ```

2. Import the `Plotly.NET` and `Plotly.NET.LayoutObjects` packages.

   ```
   using Plotly.NET;
   using Plotly.NET.LayoutObjects;
   ```

Get Historical Data

Get some historical market data to produce the plots. For example, to get data for a bank sector ETF and some banking companies over 2021, run:

```
var qb = new QuantBook();
var tickers = new[]
{
    "XLF",  // Financial Select Sector SPDR Fund
    "COF",  // Capital One Financial Corporation
    "GS",   // Goldman Sachs Group, Inc.
    "JPM",  // J P Morgan Chase & Co
    "WFC"   // Wells Fargo & Company
}
var symbols = tickers.Select(ticker => qb.AddEquity(ticker, Resolution.Daily).Symbol);
var history = qb.History(symbols, new DateTime(2021, 1, 1), new DateTime(2022, 1, 1));
```

Create Candlestick Chart

You must import the plotting libraries and get some historical data to create candlestick charts.

In this example, you create a candlestick chart that shows the open, high, low, and close prices of one of the banking securities. Follow these steps to create the candlestick chart:

1. Select a `Symbol` .

   ```
   var symbol = symbols.First();
   ```

2. Call the `Chart2D.Chart.Candlestick` constructor with the time and open, high, low, and close price `IEnumerable` .

   ```
   var bars = history.Select(slice => slice.Bars[symbol]);
   var chart = Chart2D.Chart.Candlestick<decimal, decimal, decimal, decimal, DateTime, string>(
       bars.Select(x => x.Open),
       bars.Select(x => x.High),
       bars.Select(x => x.Low),
       bars.Select(x => x.Close),
       bars.Select(x => x.EndTime)
   );
   ```

3. Call the `Layout` constructor and set the `title` , `xaxis` , and `yaxis` properties as the title and axes label objects.

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Price ($)");
   Title title = Title.init($"{symbol} OHLC");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

4. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

5. Show the plot.

   ```
   HTML(GenericChart.toChartHTML(chart));
   ```

   The Jupyter Notebook displays the candlestick chart.

Create Line Chart

You must import the plotting libraries and get some historical data to create line charts.

In this example, you create a line chart that shows the volume of a security. Follow these steps to create the chart:

1. Select a `Symbol` .

   ```
   var symbol = symbols.First();
   ```

2. Call the `Chart2D.Chart.Line` constructor with the timestamps and volumes.

   ```
   var bars = history.Select(slice => slice.Bars[symbol]);
   var chart = Chart2D.Chart.Line<DateTime, decimal, string>(
       bars.Select(x => x.EndTime),
       bars.Select(x => x.Volume)
   );
   ```

3. Create a `Layout` .

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", "Time");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", "Volume");
   Title title = Title.init($"{symbol} Volume");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

4. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

5. Show the plot.

   ```
   HTML(GenericChart.toChartHTML(chart));
   ```

   The Jupyter Notebook displays the line chart.

Create Scatter Plot

You must import the plotting libraries and get some historical data to create scatter plots.

In this example, you create a scatter plot that shows the relationship between the daily price of two securities. Follow these steps to create the scatter plot:

1. Select two `Symbol` objects.

   ```
   var symbol1 = symbols.First();
   var symbol2 = symbols.Last();
   ```

2. Call the `Chart2D.Chart.Point` constructor with the closing prices of both securities.

   ```
   var chart = Chart2D.Chart.Point<decimal, decimal, string>(
       history.Select(slice => slice.Bars[symbol1].Close),
       history.Select(slice => slice.Bars[symbol2].Close)
   );
   ```

3. Create a `Layout` .

   ```
   LinearAxis xAxis = new LinearAxis();
   xAxis.SetValue("title", $"{symbol1} Price ($)");
   LinearAxis yAxis = new LinearAxis();
   yAxis.SetValue("title", $"{symbol2} Price ($)");
   Title title = Title.init($"{symbol1} vs {symbol2}");

   Layout layout = new Layout();
   layout.SetValue("xaxis", xAxis);
   layout.SetValue("yaxis", yAxis);
   layout.SetValue("title", title);
   ```

4. Assign the `Layout` to the chart.

   ```
   chart.WithLayout(layout);
   ```

5. Show the plot.

   ```
   HTML(GenericChart.toChartHTML(chart));
   ```

   The Jupyter Notebook displays the scatter plot.

5 Indicators

Indicators let you analyze market data in an abstract form rather than in its raw form. For example, indicators like the RSI tell you, based on price and volume data, if the market is overbought or oversold. Because indicators can extract overall market trends from price data, sometimes, you may want to look for correlations between indicators and the market, instead of between raw price data and the market. To view all of the indicators and candlestick patterns we provide, see the Supported Indicators .

Data Point Indicators

Indicators that process `IndicatorDataPoint` objects

Bar Indicators

Indicators that process `Bar` objects

Trade Bar Indicators

Indicators that process `TradeBar` objects

Combining Indicators

Chain indicators together

Custom Indicators

Create your own

Custom Resolutions

Beyond the standard resolutions

See Also

Key Concepts

5.1 Data Point Indicators

Introduction

This page explains how to create, update, and visualize LEAN data-point indicators.

Create Subscriptions

You need to subscribe to some market data in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to subscribe to some market data and create an indicator in order to calculate a timeseries of indicator values. In this example, use a 20-period 2-standard-deviation `BollingerBands` indicator.

```
var bb = new BollingerBands(20, 2);
```

```
bb = BollingerBands(20, 2)
```

You can create the indicator timeseries with the `Indicator` helper method or you can manually create the timeseries.

Indicator Helper Method

To create an indicator timeseries with the helper method, call the `Indicator` method.

```
var bbIndicator = qb.Indicator(bb, symbol, 50, Resolution.Daily);
```

```
bb_dataframe = qb.Indicator(bb, symbol, 50, Resolution.Daily)
```

Manually Create the Indicator Timeseries

Follow these steps to manually create the indicator timeseries:

1. Get some historical data.

```
var history = qb.History(symbol, 70, Resolution.Daily);
```

```
history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
```

2. Create a `RollingWindow` for each attribute of the indicator to hold their values.

```
var time = new RollingWindow<DateTime>(50);
var window = new Dictionary<string, RollingWindow<decimal>>();
window["bollingerbands"] = new RollingWindow<decimal>(50);
window["lowerband"] = new RollingWindow<decimal>(50);
window["middleband"] = new RollingWindow<decimal>(50);
window["upperband"] = new RollingWindow<decimal>(50);
window["bandwidth"] = new RollingWindow<decimal>(50);
window["percentb"] = new RollingWindow<decimal>(50);
window["standarddeviation"] = new RollingWindow<decimal>(50);
window["price"] = new RollingWindow<decimal>(50);
```

```
window = {}
window['time'] = RollingWindow[DateTime](50)
window["bollingerbands"] = RollingWindow[float](50)
window["lowerband"] = RollingWindow[float](50)
window["middleband"] = RollingWindow[float](50)
window["upperband"] = RollingWindow[float](50)
window["bandwidth"] = RollingWindow[float](50)
window["percentb"] = RollingWindow[float](50)
window["standarddeviation"] = RollingWindow[float](50)
window["price"] = RollingWindow[float](50)
```

3. Attach a handler method to the indicator that updates the `RollingWindow` objects.

```
bb.Updated += (sender, updated) =>
{
    var indicator = (BollingerBands)sender;
    time.Add(updated.EndTime);
    window["bollingerbands"].Add(updated);
    window["lowerband"].Add(indicator.LowerBand);
    window["middleband"].Add(indicator.MiddleBand);
    window["upperband"].Add(indicator.UpperBand);
    window["bandwidth"].Add(indicator.BandWidth);
    window["percentb"].Add(indicator.PercentB);
    window["standarddeviation"].Add(indicator.StandardDeviation);
    window["price"].Add(indicator.Price);
};
```

```
def UpdateBollingerBandWindow(sender: object, updated: IndicatorDataPoint) -> None:
    indicator = sender
    window['time'].Add(updated.EndTime)
    window["bollingerbands"].Add(updated.Value)
    window["lowerband"].Add(indicator.LowerBand.Current.Value)
    window["middleband"].Add(indicator.MiddleBand.Current.Value)
    window["upperband"].Add(indicator.UpperBand.Current.Value)
    window["bandwidth"].Add(indicator.BandWidth.Current.Value)
    window["percentb"].Add(indicator.PercentB.Current.Value)
    window["standarddeviation"].Add(indicator.StandardDeviation.Current.Value)
    window["price"].Add(indicator.Price.Current.Value)

bb.Updated += UpdateBollingerBandWindow
```

When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow` .

4. Iterate through the historical market data and update the indicator.

```
foreach(var bar in history)
{
    bb.Update(bar.EndTime, bar.Close);
}
```

```
for bar in history:
    bb.Update(bar.EndTime, bar.Close)
```

5. Display the data.

```
Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
foreach (var i in Enumerable.Range(0, 5).Reverse())
{
    var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
    Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
}
```

6. Populate a `DataFrame` with the data in the `RollingWindow` objects.

```
bb_dataframe = pd.DataFrame(window).set_index('time')
```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot data point indicators.

You need to create an indicator timeseries to plot the indicator values.

Follow these steps to plot the indicator values:

1. Select the columns/features to plot.

   ```
   bb_plot = bb_indicator[["upperband", "middleband", "lowerband", "price"]]
   ```

2. Call the `plot` method.

   ```
   bb_plot.plot(figsize=(15, 10), title="SPY BB(20,2)"))
   ```

3. Show the plots.

   ```
   plt.show()
   ```

5.2 Bar Indicators

Introduction

This page explains how to create, update, and visualize LEAN bar indicators.

Create Subscriptions

You need to subscribe to some market data in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to subscribe to some market data and create an indicator in order to calculate a timeseries of indicator values. In this example, use a 20-period `AverageTrueRange` indicator.

```
var atr = new AverageTrueRange(20);
```

```
atr = AverageTrueRange(20)
```

You can create the indicator timeseries with the `Indicator` helper method or you can manually create the timeseries.

Indicator Helper Method

To create an indicator timeseries with the helper method, call the `Indicator` method.

```
var atrIndicator = qb.Indicator(atr, symbol, 50, Resolution.Daily);
```

```
atr_dataframe = qb.Indicator(atr, symbol, 50, Resolution.Daily)
```

Manually Create the Indicator Timeseries

Follow these steps to manually create the indicator timeseries:

1. Get some historical data .

   ```
   var history = qb.History(symbol, 70, Resolution.Daily);
   ```

   ```
   history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
   ```

2. Create a `RollingWindow` for each attribute of the indicator to hold their values.

   ```
   var time = new RollingWindow<DateTime>(50);
   var window = new Dictionary<string, RollingWindow<decimal>>();
   window["averagetruerange"] = new RollingWindow<decimal>(50);
   window["truerange"] = new RollingWindow<decimal>(50);
   ```

   ```
   window = {}
   window['time'] = RollingWindow[DateTime](50)
   window['averagetruerange'] = RollingWindow[float](50)
   window["truerange"] = RollingWindow[float](50)
   ```

3. Attach a handler method to the indicator that updates the `RollingWindow` objects.

   ```
   atr.Updated += (sender, updated) =>
   {
       var indicator = (AverageTrueRange)sender;
       time.Add(updated.EndTime);
       window["averagetruerange"].Add(updated);
       window["truerange"].Add(indicator.TrueRange);
   };
   ```

   ```
   def UpdateAverageTrueRangeWindow(sender: object, updated: IndicatorDataPoint) -> None:
       indicator = sender
       window['time'].Add(updated.EndTime)
       window["averagetruerange"].Add(updated.Value)
       window["truerange"].Add(indicator.TrueRange.Current.Value)

   atr.Updated += UpdateAverageTrueRangeWindow
   ```

   When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow` .

4. Iterate through the historical market data and update the indicator.

   ```
   foreach(var bar in history){
       // Update the indicators with the whole bar.
       atr.Update(bar);
   }
   ```

   ```
   for bar in history:
       atr.Update(bar)
   ```

5. Display the data.

   ```
   Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
   foreach (var i in Enumerable.Range(0, 5).Reverse())
   {
       var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
       Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
   }
   ```

6. Populate a `DataFrame` with the data in the `RollingWindow` objects.

   ```
   atr_dataframe = pd.DataFrame(window).set_index('time')
   ```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot bar indicators.

You need to create an indicator timeseries to plot the indicator values.

Follow these steps to plot the indicator values:

1. Call the `plot` method.

   ```
   atr_indicator.plot(title="SPY ATR(20)", figsize=(15, 10))
   ```

2. Show the plots.

   ```
   plt.show()
   ```

5.3 Trade Bar Indicators

Introduction

This page explains how to create, update, and visualize LEAN `TradeBar` indicators.

Create Subscriptions

You need to [subscribe to some market data](#) in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to [subscribe to some market data](#) and create an indicator in order to calculate a timeseries of indicator values. In this example, use a 20-period `VolumeWeightedAveragePriceIndicator` indicator.

```
var vwap = new VolumeWeightedAveragePriceIndicator(20);
```

```
vwap = VolumeWeightedAveragePriceIndicator(20)
```

You can create the indicator timeseries with the `Indicator` helper method or you can manually create the timeseries.

Indicator Helper Method

To create an indicator timeseries with the helper method, call the `Indicator` method.

```
var vwapIndicator = qb.Indicator(vwap, symbol, 50, Resolution.Daily);
```

```
vwap_dataframe = qb.Indicator(vwap, symbol, 50, Resolution.Daily)
```

Manually Create the Indicator Timeseries

Follow these steps to create an indicator timeseries:

1. Get some [historical data](#).

   ```
   var history = qb.History(symbol, 70, Resolution.Daily);
   ```

   ```
   history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
   ```

2. Create a `RollingWindow` for each attribute of the indicator to hold their values.

   ```
   var time = new RollingWindow<DateTime>(50);
   var window = new Dictionary<string, RollingWindow<decimal>>();
   window["volumeweightedaveragepriceindicator"] = new RollingWindow<decimal>(50);
   ```

   ```
   window = {}
   window['time'] = RollingWindow[DateTime](50)
   window['volumeweightedaveragepriceindicator'] = RollingWindow[float](50)
   ```

3. Attach a handler method to the indicator that updates the `RollingWindow` objects.

   ```
   vwap.Updated += (sender, updated) =>
   {
       time.Add(updated.EndTime);
       window["volumeweightedaveragepriceindicator"].Add(updated);
   };
   ```

   ```
   def UpdateVWAPWindow(sender: object, updated: IndicatorDataPoint) -> None:
       window['time'].Add(updated.EndTime)
       window["volumeweightedaveragepriceindicator"].Add(updated.Value)

   vwap.Updated += UpdateVWAPWindow
   ```

   When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow`.

4. Iterate through the historical market data and update the indicator.

   ```
   foreach(var bar in history){
       // Update the indicators with the whole TradeBar.
       vwap.Update(bar);
   }
   ```

   ```
   for bar in history:
       vwap.Update(bar)
   ```

5. Display the data.

   ```
   Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
   foreach (var i in Enumerable.Range(0, 5).Reverse())
   {
       var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
       Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
   }
   ```

6. Populate a `DataFrame` with the data in the `RollingWindow` objects.

   ```
   vwap_dataframe = pd.DataFrame(window).set_index('time')
   ```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot TradeBar indicators.

Follow these steps to plot the indicator values:

1. Call the `plot` method.

   ```
   vwap_indicator.plot(title="SPY VWAP(20)", figsize=(15, 10))
   ```

2. Show the plots.

   ```
   plt.show()
   ```

5.4 Combining Indicators

Introduction

This page explains how to create, update, and visualize LEAN Composite indicators.

Create Subscriptions

You need to subscribe to some market data in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to subscribe to some market data and create a composite indicator in order to calculate a timeseries of indicator values. In this example, use a 10-period `SimpleMovingAverage` of a 10-period `RelativeStrengthIndex` indicator.

```
var rsi = new RelativeStrengthIndex(10);
var sma = new SimpleMovingAverage(10);
var smaOfRsi = IndicatorExtensions.Of(sma, rsi);
```

```
rsi = RelativeStrengthIndex(10)
sma = SimpleMovingAverage(10)
sma_of_rsi = IndicatorExtensions.Of(sma, rsi)
```

Follow these steps to create an indicator timeseries:

1. Get some historical data .

   ```
   var history = qb.History(symbol, 70, Resolution.Daily);
   ```

   ```
   history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
   ```

2. Create a `RollingWindow` for each attribute of the indicator to hold their values.

   In this example, save 50 data points.

   ```
   var time = new RollingWindow<DateTime>(50);
   var window = new Dictionary<string, RollingWindow<decimal>>();
   window["SMA Of RSI"] = new RollingWindow<decimal>(50);
   window["rollingsum"] = new RollingWindow<decimal>(50);
   ```

   ```
   window = {}
   window['time'] = RollingWindow[DateTime](50)
   window["SMA Of RSI"] = RollingWindow[float](50)
   window["rollingsum"] = RollingWindow[float](50)(50)
   ```

3. Attach a handler method to the indicator that updates the `RollingWindow` objects.

   ```
   smaOfRsi.Updated += (sender, updated) =>
   {
       var indicator = (SimpleMovingAverage)sender;    // Use terminal indicator class.
       time.Add(updated.EndTime);
       window["SMA Of RSI"].Add(updated);
       window["rollingsum"].Add(indicator.RollingSum);
   };
   ```

   ```
   def UpdateSmaOfRsiWindow(sender: object, updated: IndicatorDataPoint) -> None:
       indicator = sender
       window['time'].Add(updated.EndTime)
       window["SMA Of RSI"].Add(updated.Value)
       window["rollingsum"].Add(indicator.RollingSum.Current.Value)

   sma_of_rsi.Updated += UpdateSmaOfRsiWindow
   ```

   When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow` .

4. Iterate the historical market data to update the indicators and the `RollingWindow` s.

   ```
   foreach(var bar in history){
       // Update the base indicators, the composite indicator will update automatically when the base indicator is updated.
       rsi.Update(bar.EndTime, bar.Close);
   }
   ```

   ```
   for bar in history:
       rsi.Update(bar.EndTime, bar.Close)
   ```

5. Display the data.

   ```
   Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
   foreach (var i in Enumerable.Range(0, 5).Reverse())
   {
       var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
       Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
   }
   ```

6. Populate a `DataFrame` with the data in the `RollingWindow` objects.

   ```
   sma_of_rsi_dataframe = pd.DataFrame(window).set_index('time')
   ```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot composite indicators.

Follow these steps to plot the indicator values:

1. Select the columns/features to plot.

   ```
   sma_of_rsi_plot = sma_of_rsi_dataframe[["SMA Of RSI"]]
   ```

2. Call the `plot` method.

   ```
   sma_of_rsi_plot.plot(title="SPY SMA(10) of RSI(10)", figsize=(15, 10))
   ```

3. Show the plots.

   ```
   plt.show()
   ```

5.5 Custom Indicators

Introduction

This page explains how to create and update custom indicators.

Create Subscriptions

You need to subscribe to some market data in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to subscribe to some market data in order to calculate a timeseries of indicator values.

Follow these steps to create an indicator timeseries:

1. Get some historical data .

   ```
   var history = qb.History(symbol, 70, Resolution.Daily);
   ```

   ```
   history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
   ```

2. Define a custom indicator class that inherits from the `Indicator` superclass.
3. Define a custom indicator class. Note the `PythonIndicator` superclass inheritance, `Value` attribute, and `Update` method are mandatory.

   In this tutorial, create an `ExpectedShortfallPercent` indicator that uses Monte Carlo to calculate the expected shortfall of returns. Use the `WindowIndicator` superclass instead of `Indicator` for using a period of historical data stored in a `RollingWindow` .

   In this tutorial, create an `ExpectedShortfallPercent` indicator that uses Monte Carlo to calculate the expected shortfall of returns.

   ```
   public class ExpectedShortfallPercent : WindowIndicator<IndicatorDataPoint>, IIndicatorWarmUpPeriodProvider
   {
       private decimal _alpha;

       // Set up IndicatorDataPoint attributes for the indicator.
       public IndicatorBase<IndicatorDataPoint> ValueAtRisk { get; }

       // Set up the WarmUpPeriod attribute to provide implementation of the IIndicatorWarmUpPeriodProvider interface.
       public override int WarmUpPeriod => Period;

       // Set up the constructor.
       // period: The lookback period for return distribution.
       // alpha: Alpha level of VaR cutoff.
       public ExpectedShortfallPercent(int period, decimal alpha)
           : base("ES", period)
       {
           _alpha = alpha;
           ValueAtRisk = new Identity("ES_VaR");
       }

       // Override the IsReady method to set up the flag of the Indicator and its IndicatorDataPoint attributes are ready.
       public override bool IsReady => ValueAtRisk.IsReady;

       // Mandatory: Override the ComputeNextValue method to calculate the indictor value.
       protected override decimal ComputeNextValue(IReadOnlyWindow<IndicatorDataPoint> window, IndicatorDataPoint input)
       {
           if (Samples < 2)
               return 0m;

           var n = Math.Min(Period, Samples);
           var cutoff = (int) Math.Ceiling(n * _alpha);

           var samples = new List<decimal>();
           for (int i = 0; i < window.Count - 1; i++)
           {
               samples.Add( (window[i] - window[i+1]) / window[i+1] );
           }
           var lowest = samples.OrderBy(x => x).Take(cutoff);

           ValueAtRisk.Update(input.Time, lowest.Last());
           return lowest.Average();
       }
   }
   ```

   ```
   class ExpectedShortfallPercent(PythonIndicator):
       import math, numpy as np

       def __init__(self, period, alpha):
           self.Value = None   # Attribute represents the indicator value
           self.ValueAtRisk = None

           self.alpha = alpha

           self.window = RollingWindow[float](period)

       # Override the IsReady attribute to flag all attributes values are ready.
       @property
       def IsReady(self) -> bool:
           return self.Value and self.ValueAtRisk

       # Method to update the indicator values. Note that it only receives 1 IBaseData object (Tick, TradeBar, QuoteBar) argument.
       def Update(self, input: BaseData) -> bool:
           count = self.window.Count

           self.window.Add(input.Close)

           # Update the Value and other attributes as the indicator current value.
           if count >= 2:
               cutoff = math.ceil(self.alpha * count)

               ret = [ (self.window[i] - self.window[i+1]) / self.window[i+1] for i in range(count-1) ]
               lowest = sorted(ret)[:cutoff]

               self.Value = np.mean(lowest)
               self.ValueAtRisk = lowest[-1]

           # return a boolean to indicate IsReady.
           return count >= 2
   ```

4. Initialize a new instance of the custom indicator.

   ```
   var es = new ExpectedShortfallPercent(50, 0.05m);
   ```

   ```
   custom = ExpectedShortfallPercent(50, 0.05)
   ```

5. Create a `RollingWindow` for each attribute of the indicator to hold their values.

   In this example, save 20 data points.

   ```
   var time = new RollingWindow<DateTime>(20);
   ```

```
var window = new Dictionary<string, RollingWindow<decimal>>();
window["expectedshortfall"] = new RollingWindow<decimal>(20);
window["valueatrisk"] = new RollingWindow<decimal>(20);
```

```
window = {}
window['time'] = RollingWindow[DateTime](20)
window['expectedshortfall'] = RollingWindow[float](20)
window['valueatrisk'] = RollingWindow[float](20)
```

6. Attach a handler method to the indicator that updates the `RollingWindow` objects.

```
es.Updated += (sender, updated) =>
{
    var indicator = (ExpectedShortfallPercent) sender;
    time.Add(updated.EndTime);
    window["expectedshortfall"].Add(updated);
    window["valueatrisk"].Add(indicator.ValueAtRisk.Current);
};
```

When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow` .

7. Iterate through the historical market data and update the indicator.

```
foreach(var bar in history){
    es.Update(bar.EndTime, bar.Close);
}
```

```
for bar in history:
    custom.Update(bar)

    # The Updated event handler is not available for custom indicator in Python, RollingWindows are needed to be updated in here.
    if custom.IsReady:
        window['time'].Add(bar.EndTime)
        window['expectedshortfall'].Add(custom.Value)
        window['valueatrisk'].Add(custom.ValueAtRisk)
```

8. Display the data.

```
Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
foreach (var i in Enumerable.Range(0, 5).Reverse())
{
    var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
    Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
}
```

9. Populate a `DataFrame` with the data in the `RollingWindow` objects.

```
custom_dataframe = pd.DataFrame(window).set_index('time'))
```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot custom indicators.

Follow these steps to plot the indicator values:

1. Call the `plot` method.

```
custom_dataframe.plot()
```

2. Show the plot.

```
plt.show()
```

5.6 Custom Resolutions

Introduction

This page explains how to create and update indicators with data of a custom resolution.

Create Subscriptions

You need to subscribe to some market data in order to calculate indicator values.

```
var qb = new QuantBook();
var symbol = qb.AddEquity("SPY").Symbol;
```

```
qb = QuantBook()
symbol = qb.AddEquity("SPY").Symbol
```

Create Indicator Timeseries

You need to subscribe to some market data and create an indicator in order to calculate a timeseries of indicator values.

Follow these steps to create an indicator timeseries:

1. Get some historical data .

   ```
   var history = qb.History(symbol, 70, Resolution.Daily);
   ```

   ```
   history = qb.History[TradeBar](symbol, 70, Resolution.Daily)
   ```

2. Create a data-point indicator.

   In this example, use a 20-period 2-standard-deviation `BollingerBands` indicator.

   ```
   var bb = new BollingerBands(20, 2);
   ```

   ```
   bb = BollingerBands(20, 2)
   ```

3. Create a `RollingWindow` for each attribute of the indicator to hold their values.

   ```
   var time = new RollingWindow<DateTime>(50);
   var window = new Dictionary<string, RollingWindow<decimal>>();
   window["bollingerbands"] = new RollingWindow<decimal>(50);
   window["lowerband"] = new RollingWindow<decimal>(50);
   window["middleband"] = new RollingWindow<decimal>(50);
   window["upperband"] = new RollingWindow<decimal>(50);
   window["bandwidth"] = new RollingWindow<decimal>(50);
   window["percentb"] = new RollingWindow<decimal>(50);
   window["standarddeviation"] = new RollingWindow<decimal>(50);
   window["price"] = new RollingWindow<decimal>(50);
   ```

   ```
   window = {}
   window['time'] = RollingWindow[DateTime](50)
   window["bollingerbands"] = RollingWindow[float](50)
   window["lowerband"] = RollingWindow[float](50)
   window["middleband"] = RollingWindow[float](50)
   window["upperband"] = RollingWindow[float](50)
   window["bandwidth"] = RollingWindow[float](50)
   window["percentb"] = RollingWindow[float](50)
   window["standarddeviation"] = RollingWindow[float](50)
   window["price"] = RollingWindow[float](50)
   ```

4. Attach a handler method to the indicator that updates the `RollingWindow` objects.

   ```
   bb.Updated += (sender, updated) =>
   {
       var indicator = (BollingerBands)sender;
       time.Add(updated.EndTime);
       window["bollingerbands"].Add(updated);
       window["lowerband"].Add(indicator.LowerBand);
       window["middleband"].Add(indicator.MiddleBand);
       window["upperband"].Add(indicator.UpperBand);
       window["bandwidth"].Add(indicator.BandWidth);
       window["percentb"].Add(indicator.PercentB);
       window["standarddeviation"].Add(indicator.StandardDeviation);
       window["price"].Add(indicator.Price);
   };
   ```

   ```
   def UpdateBollingerBandWindow(sender: object, updated: IndicatorDataPoint) -> None:
       indicator = sender
       window['time'].Add(updated.EndTime)
       window["bollingerbands"].Add(updated.Value)
       window["lowerband"].Add(indicator.LowerBand.Current.Value)
       window["middleband"].Add(indicator.MiddleBand.Current.Value)
       window["upperband"].Add(indicator.UpperBand.Current.Value)
       window["bandwidth"].Add(indicator.BandWidth.Current.Value)
       window["percentb"].Add(indicator.PercentB.Current.Value)
       window["standarddeviation"].Add(indicator.StandardDeviation.Current.Value)
       window["price"].Add(indicator.Price.Current.Value)

   bb.Updated += UpdateBollingerBandWindow
   ```

   When the indicator receives new data, the preceding handler method adds the new `IndicatorDataPoint` values into the respective `RollingWindow` .

5. Create a `TradeBarConsolidator` to consolidate data into the custom resolution.

   ```
   var consolidator = new TradeBarConsolidator(TimeSpan.FromDays(7));
   ```

   ```
   consolidator = TradeBarConsolidator(timedelta(days=7))
   ```

6. Attach a handler method to feed data into the consolidator and updates the indicator with the consolidated bars.

   ```
   consolidator.DataConsolidated += (sender, consolidated) =>
   {
       bb.Update(consolidated.EndTime, consolidated.Close);
   };
   ```

   ```
   consolidator.DataConsolidated += lambda sender, consolidated: bb.Update(consolidated.EndTime, consolidated.Close)
   ```

   When the consolidator receives 7 days of data, the handler generates a 7-day `TradeBar` and update the indicator.

7. Iterate through the historical market data and update the indicator.

   ```
   foreach(var bar in history)
   {
       consolidator.Update(bar);
   }
   ```

   ```
   for bar in history:
       consolidator.Update(bar)
   ```

8. Display the data.

   ```
   Console.WriteLine($"time,{string.Join(',', window.Select(kvp => kvp.Key))}");
   foreach (var i in Enumerable.Range(0, 5).Reverse())
   {
       var data = string.Join(", ", window.Select(kvp => Math.Round(kvp.Value[i],6)));
       Console.WriteLine($"{time[i]:yyyyMMdd}, {data}");
   }
   ```

9. Populate a `DataFrame` with the data in the `RollingWindow` objects.

   ```
   bb_dataframe = pd.DataFrame(window).set_index('time')
   ```

Plot Indicators

Jupyter Notebooks don't currently support libraries to plot historical data, but we are working on adding the functionality. Until the functionality is added, use Python to plot indicators.

Follow these steps to plot the indicator values:

1. Select the columsn to plot.

   ```
   df = bb_dataframe[['lowerband', 'middleband', 'upperband', 'price']]
   ```

2. Call the `plot` method.

   ```
   df.plot()
   ```

3. Show the plot.

   ```
   plt.show()
   ```

6 Object Store

Introduction

The Object Store is a file system that you can use in your algorithms to save, read, and delete data. The Object Store is organization-specific, so you can save or read data from the same Object Store in all of your organization's projects. The Object Store works like a key-value storage system where you can store regular strings, JSON encoded strings, XML encoded strings, and bytes. You can access the data you store in the Object Store from backtests, the Research Environment, and live algorithms.

When you deploy live algorithms, the state of the Object Store is copied, but it never refreshes. Therefore, if you save data in the Object Store in a live algorithm, you can access the data from the live algorithm, backtests, and the Research Environment. However, if you save content into the Object Store from the Research Environment or a backtest after you deploy a live algorithm, you can't access the new content from the live algorithm.

Get All Stored Data

To get all of the keys and values in the Object Store, iterate through the `ObjectStore` object.

```
foreach (var kvp in qb.ObjectStore)
{
    var key = kvp.Key;
    var value = kvp.Value;
}
```

```
for kvp in qb.ObjectStore:
    key = kvp.Key
    value = kvp.Value
```

To iterate through just the keys in the Object Store, iterate through the `Keys` property.

```
foreach (var key in qb.ObjectStore.Keys)
{
    continue;
}
```

```
for key in qb.ObjectStore.Keys:
    continue
```

Create Sample Data

You need some data to store data in the Object Store.

Follow these steps to create some sample data:

1. Create a dictionary.

   ```
   var dictSample = new Dictionary<string, int> { {"One", 1}, {"Two", 2}, {"Three", 3} };
   ```

2. Create a `string` .

   ```
   var stringSample = "My string";
   ```

   ```
   string_sample = "My string"
   ```

3. Create a `Bytes` object.

   ```
   var bytesSample = Encoding.UTF8.GetBytes("My String");
   ```

   ```
   bytes_sample = str.encode("My String")
   ```

4. Convert the dictionary to an `XML` -formatted object.

   ```
   var xmlSample = new XElement("sample",
       dictSample.Select(kvp => new XElement(kvp.Key, kvp.Value)));
   Console.WriteLine(xmlSample.ToString());
   ```

Save Data

The Object Store saves objects under a key-value system. If you save objects in backtests, you can access them from the Research Environment.

If you run algorithms in QuantConnect Cloud, you need storage create permissions to save data in the Object Store.

If you don't have data to store, create some sample data .

You can save the following types of objects in the Object Store:

- `Bytes` objects
- `string` objects
- JSON objects
- XML-formatted objects

You can save `Bytes` and `string` objects in the Object Store.

Bytes

To save a `Bytes` object, call the `SaveBytes` method.

```
var saveSuccessful = qb.ObjectStore.SaveBytes($"{qb.ProjectId}/bytesKey", bytesSample)
```

```
save_successful = qb.ObjectStore.SaveBytes(f"{qb.ProjectId}/bytes_key", bytes_sample)
```

Strings

To save a `string` object, call the `Save` or `SaveString` method.

```
var saveSuccessful = qb.ObjectStore.Save($"{qb.ProjectId}/stringKey", stringSample);
```

```
save_successful = qb.ObjectStore.Save(f"{qb.ProjectId}/string_key", string_sample)
```

JSON

To save a JSON object, call the `SaveJson<T>` method. This method helps to serialize the data into JSON format.

```
var saveSuccessful = qb.ObjectStore.SaveJson<Dictionary<string, int>>($"{qb.ProjectId}/jsonKey", dictSample);
```

XML

To save an XML-formatted object, call the `SaveXml<T>` method.

```
var saveSuccessful = qb.ObjectStore.SaveXml<XElement>($"{qb.ProjectId}/xmlKey", xmlSample);
```

Read Data

To read data from the Object Store, you need to provide the key you used to store the object.

You can load the following types of objects from the Object Store:

- `Bytes` objects
- `string` objects

- JSON objects
- XML-formatted objects

You can load `Bytes` and `string` objects from the Object Store.

Before you read data from the Object Store, check if the key exists.

```
if (qb.ObjectStore.ContainsKey(key))
{
    // Read data
}
```

```
if qb.ObjectStore.ContainsKey(key):
    # Read data
```

Bytes

To read a `Bytes` object, call the `ReadBytes` method.

```
var bytesData = qb.ObjectStore.ReadBytes($"{qb.ProjectId}/bytesKey");
```

```
byte_data = qb.ObjectStore.ReadBytes(f"{qb.ProjectId}/bytes_key")
```

Strings

To read a `string` object, call the `Read` or `ReadString` method.

```
var stringData = qb.ObjectStore.Read($"{qb.ProjectId}/stringKey");
```

```
string_data = qb.ObjectStore.Read(f"{qb.ProjectId}/string_key")
```

JSON

To read a JSON object, call the `ReadJson<T>` method.

```
var jsonData = qb.ObjectStore.ReadJson<Dictionary<string, int>>($"{qb.ProjectId}/jsonKey");
```

XML

To read an XML-formatted object, call the `ReadXml<T>` method.

```
var xmlData = qb.ObjectStore.ReadXml<XElement>($"{qb.ProjectId}/xmlKey");
```

If you created the XML object from a dictionary, reconstruct the dictionary.

```
var dict = xmlData.Elements().ToDictionary(x => x.Name.LocalName, x => int.Parse(x.Value));
```

Delete Data

Delete objects in the Object Store to remove objects that you no longer need. If you use the Research Environment in QuantConnect Cloud, you need [storage delete permissions](#) to delete data from the Object Store.

To delete objects from the Object Store, call the `Delete` method. Before you delete data, check if the key exists. If you try to delete an object with a key that doesn't exist in the Object Store, the method raises an exception.

```
if (qb.ObjectStore.ContainsKey(key))
{
    qb.ObjectStore.Delete(key);
}
```

```
if qb.ObjectStore.ContainsKey(key):
    qb.ObjectStore.Delete(key)
```

To delete all of the content in the Object Store, iterate through all the stored data.

```
foreach (var kvp in qb.ObjectStore)
{
    qb.ObjectStore.Delete(kvp.Key);
}
```

```
for kvp in qb.ObjectStore:
    qb.ObjectStore.Delete(kvp.Key)
```

Cache Data

When you write to or read from the Object Store, the notebook caches the data. The cache speeds up the notebook execution because if you try to read the Object Store data again with the same key, it returns the cached data instead of downloading the data again. The cache speeds up execution, but it can cause problems if you are trying to share data between two nodes under the same Object Store key. For example, consider the following scenario:

1. You open project A and save data under the key `123` .
2. You open project B and save new data under the same key `123` .
3. In project A, you read the Object Store data under the key `123` , expecting the data from project B, but you get the original data you saved in step #1 instead.

   You get the data from step 1 instead of step 2 because the cache contains the data from step 1.

To clear the cache, call the `Clear` method.

```
qb.ObjectStore.Clear();
```

```
qb.ObjectStore.Clear()
```

Get File Path

To get the file path for a specific key in the Object Store, call the `GetFilePath` method. If the key you pass to the method doesn't already exist in the Object Store, it's added to the Object Store.

```
var filePath = qb.ObjectStore.GetFilePath(key);
```

```
file_path = qb.ObjectStore.GetFilePath(key)
```

Storage Quotas

If you use the Research Environment locally, you can store as much data as your hardware will allow. If you use the Research Environment in QuantConnect Cloud, you must stay within your [storage quota](#) . If you need more storage space, [edit your storage plan](#) .

Example

You can use the `ObjectStore` to plot data from your backtests and live algorithm in the Research Environment. In the following example, you will learn how to plot the [Simple Moving Average](#) indicator generated in a backtest.

1. Create a algorithm, add a data subscription and a [Simple Moving Average](#) indicator.

   ```
   public class ObjectStoreChartingAlgorithm : QCAlgorithm
   {
       private SimpleMovingAverage _sma;
       private string _content;

       public override void Initialize()
       {
           AddEquity("SPY", Resolution.Minute);
           _sma = SMA("SPY", 22);
       }
   }
   ```

```
class ObjectStoreChartingAlgorithm(QCAlgorithm):
    def Initialize(self):
        self.AddEquity("SPY")

        self.content = ''
        self.sma = self.SMA("SPY", 22)
```

The algorithm will save `_content` `self.content` to the `ObjectStore` .

2. Save indicator data as `string` in `_content` `self.content` .

```
public override void OnData(Slice data)
{
    _content += $"{_sma.Current.EndTime},{_sma}\n";
}
```

```
def OnData(self, data: Slice):
    self.Plot('SMA', 'Value', self.sma.Current.Value)
    self.content += f'{self.sma.Current.EndTime},{self.sma.Current.Value}\n'
```

3. To store the collected data, call the `Save` method with a key.

```
public override void OnEndOfAlgorithm()
{
    ObjectStore.Save("sma_values_csharp", _content);
}
```

```
def OnEndOfAlgorithm(self):
    self.ObjectStore.Save('sma_values_python', self.content)
```

4. Open the Research Environment, and create a `QuantBook` .

```
// Execute the following command in first
#load "../Initialize.csx"

// Create a QuantBook object
#load "../QuantConnect.csx"
using QuantConnect;
using QuantConnect.Research;

var qb = new QuantBook();
```

```
qb = QuantBook()
```

5. To read data from the Object Store, call the `Read` method. You need to provide the key you used to store the object.

```
var content = qb.ObjectStore.Read("sma_values_csharp");
```

```
content = qb.ObjectStore.Read("sma_values_python")
```

6. Convert the data to a pandas object, and create a chart.

```
data = {}
for line in content.split('\n'):
    csv = line.split(',')
    if len(csv) > 1:
        data[csv[0]] = float(csv[1])

series = pd.Series(data, index=data.keys())
series.plot()
```

7. Import the `Plotly.NET` and `Plotly.NET.LayoutObjects` packages.

```
#r "../Plotly.NET.dll"
using Plotly.NET;
using Plotly.NET.LayoutObjects;
```

8. Create the `Layout` object, and set the `title` , `xaxis` , and `yaxis` properties.

```
var layout = new Layout();
layout.SetValue("title", Title.init("SMA"));

var xAxis = new LinearAxis();
xAxis.SetValue("title", "Time");
layout.SetValue("xaxis", xAxis);

var yAxis = new LinearAxis();
yAxis.SetValue("title", "SMA");
layout.SetValue("yaxis", yAxis);
```

9. Convert the data to a list of `DateTime` objects for the chart x-axis and a list of `decimal` objects for the chart y-axis. Create a `Chart2D.Chart.Line` object with the data.

```
var index = new List<DateTimee>();
var values = new List<decimal>();

foreach (var line in content.Split('\n'))
{
    var csv = line.Split(',');
    if (csv.Length > 1)
    {
        index.Add(Parse.DateTime(csv[0]));
        values.Add(decimal.Parse(csv[1]));
    }
}

var chart = Chart2D.Chart.Linee<DateTime, decimal, stringe>(index, values);
```

10. Apply the layout to the `Line` object, and create the `HTML` object.

```
chart.WithLayout(layout);
var result = HTML(GenericChart.toChartHTML(chart));
```

- Charts
- Statistics
- Code
  - Main.cs
  - Research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	0
Average Win	0%
Average Loss	0%
Compounding Annual Return	0%
Drawdown	0%
Expectancy	0
Net Profit	0%
Sharpe Ratio	0
Probabilistic Sharpe Ratio	0%
Loss Rate	0%
Win Rate	0%
Profit-Loss Ratio	0
Alpha	0
Beta	0
Annual Standard Deviation	0
Annual Variance	0
Information Ratio	-1.692
Tracking Error	0.138
Treynor Ratio	0
Total Fees	$0.00
Estimated Strategy Capacity	$0
Lowest Capacity Asset	

```
#region imports
    using QuantConnect.Indicators;
    using QuantConnect.Data;
    using QuantConnect.Storage;
#endregion
namespace QuantConnect.Algorithm.CSharp
{
    public class ObjectStoreChartingAlgorithm : QCAlgorithm
    {
        private SimpleMovingAverage _sma;
```

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	0
Average Win	0%
Average Loss	0%
Compounding Annual Return	0%
Drawdown	0%
Expectancy	0
Net Profit	0%
Sharpe Ratio	0
Probabilistic Sharpe Ratio	0%
Loss Rate	0%
Win Rate	0%
Profit-Loss Ratio	0
Alpha	0
Beta	0
Annual Standard Deviation	0
Annual Variance	0
Information Ratio	-1.692
Tracking Error	0.138
Treynor Ratio	0
Total Fees	$0.00
Estimated Strategy Capacity	$0
Lowest Capacity Asset	

```
# region imports
from AlgorithmImports import *
# endregion

class ObjectStoreChartingAlgorithm(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2023, 1, 1)  # Set Start Date
        self.SetCash(100000)           # Set Strategy Cash
        self.AddEquity("SPY", Resolution.Minute)
```

7 Machine Learning

7.1 Key Concepts

Introduction

Machine learning is a field of study that combines statistics and computer science to build intelligent systems that predict outcomes. Quant researchers commonly use machine learning models to optimize portfolios, make trading signals, and manage risk. These models can find relationships in datasets that humans struggle to find, are subtle, or are too complex. You can use machine learning techniques in your research notebooks.

Supported Libraries

The following table shows the supported machine learning libraries:

Library	Research Tutorial	Documentation
Keras	Tutorial	Documentation
TensorFlow	Tutorial	Documentation
Scikit-Learn	Tutorial	Documentation
hmmlearn	Tutorial	Documentation
gplearn	Tutorial	Documentation
PyTorch	Tutorial	Documentation
Stable Baselines	Tutorial	Documentation
tslearn	Tutorial	Documentation
XGBoost	Tutorial	Documentation

Add New Libraries

To request a new library, contact us . We will add the library to the queue for review and deployment. Since the libraries run on our servers, we need to ensure they are secure and won't cause harm. The process of adding new libraries takes 2-4 weeks to complete. View the list of libraries currently under review on the Issues list of the Lean GitHub repository .

Transfer Models

You can load machine learning models from the Object Store or a custom data file like pickle. If you train a model in the Research Environment, you can also save it into the Object Store to transfer it to the backtesting and live trading environment.

7.2 Keras

Introduction

This page explains how to build, train, test, and store `keras` models.

Import Libraries

Import the `keras` libraries.

```
from tensorflow.keras import utils, models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import RMSprop
```

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	Daily percent change of the open, high, low, close, and volume of the SPY over the last 5 days
Labels	Daily percent return of the SPY over the next day

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Call the `pct_change` and `dropna` methods.

   ```
   daily_pct_change = history.pct_change().dropna()
   ```

2. Loop through the `daily_pct_change` DataFrame and collect the features and labels.

   ```
   n_steps = 5
   features = []
   labels = []
   for i in range(len(daily_pct_change)-n_steps):
       features.append(daily_pct_change.iloc[i:i+n_steps].values)
       labels.append(daily_pct_change['close'].iloc[i+n_steps])
   ```

3. Convert the lists of features and labels into `numpy` arrays.

   ```
   features = np.array(features)
   labels = np.array(labels)
   ```

4. Split the data into training and testing periods.

   ```
   train_length = int(len(features) * 0.7)
   X_train = features[:train_length]
   X_test = features[train_length:]
   y_train = labels[:train_length]
   y_test = labels[train_length:]
   ```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, build a neural network model that predicts the future return of the SPY. Follow these steps to create the model:

1. Call the `Sequential` constructor with a list of layers.

   ```
   model = Sequential([Dense(10, input_shape=(5,5), activation='relu'),
                       Dense(10, activation='relu'),
                       Flatten(),
                       Dense(1)])
   ```

   Set the `input_shape` of the first layer to `(5, 5)` because each sample contains the percent change of 5 factors (percent change of the open, high, low, close, and volume) over the previous 5 days. Call the `Flatten` constructor because the input is 2-dimensional but the output is just a single value.

2. Call the `compile` method with a loss function, an optimizer, and a list of metrics to monitor.

   ```
   model.compile(loss='mse',
                 optimizer=RMSprop(0.001),
                 metrics=['mae', 'mse'])
   ```

3. Call the `fit` method with the features and labels of the training dataset and a number of epochs.

   ```
   model.fit(X_train, y_train, epochs=5)
   ```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Call the `predict` method with the features of the testing period.

   ```
   y_hat = model.predict(X_test)
   ```

2. Plot the actual and predicted labels of the testing period.

   ```
   results = pd.DataFrame({'y': y_test.flatten(), 'y_hat': y_hat.flatten()})
   df.plot(title='Model Performance: predicted vs actual %change in closing price')
   ```

Store Models

You can save and load `keras` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

This method returns the file path where the model will be stored.

3. Call the `save` method the file path.

```
model.save(file_name)
```

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method with the model key.

```
qb.ObjectStore.ContainsKey(model_key)
```

This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the key name.

```
file_name = qb.ObjectStore.GetFilePath(model_key)
```

This method returns the path where the model is stored.

3. Call the `load_model` method with the file path.

```
loaded_model = load_model(file_name)
```

This method returns the saved model.

7.3 TensorFlow

Introduction

This page explains how to build, train, test, and store `Tensorflow` models.

Import Libraries

Import the `tensorflow` , `sklearn` , `json5` and `google.protobuf` libraries.

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
import json5
from google.protobuf import json_format
```

You need the `sklearn` library to prepare the data and the `json5` and `google.protobuf` libraries to save models.

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	The last 5 closing prices
Labels	The following day's closing price

Follow these steps to prepare the data:

1. Loop through the DataFrame of historical prices and collect the features.

   ```
   lookback = 5
   lookback_series = []
   for i in range(1, lookback + 1):
       df = history['close'].shift(i)[lookback:-1]
       df.name = f"close_-{i}"
       lookback_series.append(df)
   X = pd.concat(lookback_series, axis=1).reset_index(drop=True)
   ```

   The following image shows the format of the features DataFrame:

2. Select the close column and then call the `shift` method to collect the labels.

   ```
   Y = history['close'].shift(-1)
   ```

3. Drop the first 5 features and then call the `reset_index` method.

   ```
   Y = Y[lookback:-1].reset_index(drop=True)
   ```

   This method aligns the history of the features and labels.

4. Call the `train_text_split` method with the datasets and a split size.

   For example, to use the last third of data to test the model, run:

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, shuffle=False)
   ```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, build a neural network model that predicts the future price of the SPY.

Build the Model

Follow these steps to build the model:

1. Call the `reset_default_graph` method.

   ```
   tf.reset_default_graph()
   ```

   This method clears the default graph stack and resets the global default graph.

2. Call the `Session` constructor.

   ```
   sess = tf.Session()
   ```

3. Declare the number of factors and then create placeholders for the input and output layers.

   ```
   num_factors = X_test.shape[1]
   X = tf.placeholder(dtype=tf.float32, shape=[None, num_factors], name='X')
   Y = tf.placeholder(dtype=tf.float32, shape=[None])
   ```

4. Set up the weights and bias initializers for each layer.

   ```
   weight_initializer = tf.variance_scaling_initializer(mode="fan_avg", distribution="uniform", scale=1)
   bias_initializer = tf.zeros_initializer()
   ```

5. Create hidden layers that use the Relu activator.

   ```
   num_neurons_1 = 32
   num_neurons_2 = 16
   num_neurons_3 = 8

   W_hidden_1 = tf.Variable(weight_initializer([num_factors, num_neurons_1]))
   bias_hidden_1 = tf.Variable(bias_initializer([num_neurons_1]))
   hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))

   W_hidden_2 = tf.Variable(weight_initializer([num_neurons_1, num_neurons_2]))
   bias_hidden_2 = tf.Variable(bias_initializer([num_neurons_2]))
   hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2), bias_hidden_2))

   W_hidden_3 = tf.Variable(weight_initializer([num_neurons_2, num_neurons_3]))
   bias_hidden_3 = tf.Variable(bias_initializer([num_neurons_3]))
   hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3), bias_hidden_3))
   ```

6. Create the output layer and give it a name.

   ```
   W_out = tf.Variable(weight_initializer([num_neurons_3, 1]))
   bias_out = tf.Variable(bias_initializer([1]))
   output = tf.transpose(tf.add(tf.matmul(hidden_3, W_out), bias_out), name='outer')
   ```

   This snippet creates a 1-node output for both weight and bias. You must name the output layer so you can access it after you load and save the model.

7. Set up the loss function and optimizers for gradient descent optimization and backpropagation.

```
loss = tf.reduce_mean(tf.squared_difference(output, Y))
optimizer = tf.train.AdamOptimizer().minimize(loss)
```

Use mean-square error as the loss function because the close price is a continuous data and use Adam as the optimizer because of its adaptive step size.

8. Set the batch size and number of epochs to bootstrap the training process.

```
batch_size = len(y_train) // 10
epochs = 20
```

Train the Model

Follow these steps to train the model:

1. Call the `run` method with the result from the `global_variables_initializer` method.

```
sess.run(tf.global_variables_initializer())
```

2. Loop through the number of epochs, select a subset of the training data, and then call the `run` method with the subset of data.

```
for _ in range(epochs):
    for i in range(0, len(y_train) // batch_size):
        start = i * batch_size
        batch_x = X_train[start:start + batch_size]
        batch_y = y_train[start:start + batch_size]
        sess.run(optimizer, feed_dict={X: batch_x, Y: batch_y})
```

Test Models

To test the model, we'll setup a method to plot test set predictions ontop of the SPY price.

```
def test_model(sess, output, title, X):
    prediction = sess.run(output, feed_dict={X: X_test})
    prediction = prediction.reshape(prediction.shape[1], 1)

    y_test.reset_index(drop=True).plot(figsize=(16, 6), label="Actual")
    plt.plot(prediction, label="Prediction")
    plt.title(title)
    plt.xlabel("Time step")
    plt.ylabel("SPY Price")
    plt.legend()
    plt.show()

test_model(sess, output, "Test Set Results from Original Model", X)
```

Store Models

You can save and load `TensorFlow` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Export the `TensorFlow` graph as a JSON object.

```
graph_definition = tf.compat.v1.train.export_meta_graph()
json_graph = json_format.MessageToJson(graph_definition)
```

2. Export the `TensorFlow` weights as a JSON object.

```
# Define a function to get the weights from the tensorflow session
def get_json_weights(sess):
    weights = sess.run(tf.compat.v1.trainable_variables())
    weights = [w.tolist() for w in weights]
    weights_list = json5.dumps(weights)
    return weights_list

json_weights = get_json_weights(sess)
sess.close()    # Close the session opened by the `get_json_weights` function
```

3. Save the graph and weights to the `ObjectStore` .

```
qb.ObjectStore.Save('graph', json_graph)
qb.ObjectStore.Save('weights', json_weights)
```

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Read the model graph and weights from the `ObjectStore` .

```
json_graph = qb.ObjectStore.Read('graph')
json_weights = qb.ObjectStore.Read('weights')
```

2. Restore the `TensorFlow` graph from the JSON object.

```
tf.reset_default_graph()
graph_definition = json_format.Parse(json_graph, tf.compat.v1.MetaGraphDef())
sess = tf.Session()
tf.compat.v1.train.import_meta_graph(graph_definition)
```

3. Select the input and output tensors.

```
X = tf.compat.v1.get_default_graph().get_tensor_by_name('X:0')
output = tf.compat.v1.get_default_graph().get_tensor_by_name('outer:0')
```

4. Restore the model weights from the JSON object.

```
weights = [np.asarray(x) for x in json5.loads(json_weights)]
assign_ops = []
feed_dict = {}
vs = tf.compat.v1.trainable_variables()
zipped_values = zip(vs, weights)
for var, value in zipped_values:
    value = np.asarray(value)
    assign_placeholder = tf.placeholder(var.dtype, shape=value.shape)
    assign_op = var.assign(assign_placeholder)
    assign_ops.append(assign_op)
    feed_dict[assign_placeholder] = value
sess.run(assign_ops, feed_dict=feed_dict)
```

7.4 Scikit-Learn

Introduction

This page explains how to build, train, test, and store `Scikit-Learn` / `sklearn` models.

Import Libraries

Import the `sklearn` libraries.

```
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
import joblib
```

You need the `joblib` library to store models.

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	Daily percent change of the open, high, low, close, and volume of the SPY over the last 5 days
Labels	Daily percent return of the SPY over the next day

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Call the `pct_change` method and then drop the first row.

   ```
   daily_returns = history['close'].pct_change()[1:]
   ```

2. Loop through the `daily_returns` DataFrame and collect the features and labels.

   ```
   n_steps = 5
   features = []
   labels = []
   for i in range(len(daily_returns)-n_steps):
       features.append(daily_returns.iloc[i:i+n_steps].values)
       labels.append(daily_returns.iloc[i+n_steps])
   ```

3. Convert the lists of features and labels into `numpy` arrays.

   ```
   X = np.array(features)
   y = np.array(labels)
   ```

4. Split the data into training and testing periods.

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, y)
   ```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, build a Support Vector Regressor model and optimize its hyperparameters with grid search cross-validation. Follow these steps to create the model:

1. Set the choices of hyperparameters used for grid search testing.

   ```
   param_grid = {'C': [.05, .1, .5, 1, 5, 10],
                 'epsilon': [0.001, 0.005, 0.01, 0.05, 0.1],
                 'gamma': ['auto', 'scale']}
   ```

2. Call the `GridSearchCV` constructor with the SVR model, the parameter grid, a scoring method, the number of cross-validation folds.

   ```
   gsc = GridSearchCV(SVR(), param_grid, scoring='neg_mean_squared_error', cv=5)
   ```

3. Call the `fit` method and then select the best estimator.

   ```
   model = gsc.fit(X_train, y_train).best_estimator_
   ```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Call the `predict` method with the features of the testing period.

   ```
   y_hat = model.predict(X_test)
   ```

2. Plot the actual and predicted labels of the testing period.

   ```
   df = pd.DataFrame({'y': y_test.flatten(), 'y_hat': y_hat.flatten()})
   df.plot(title='Model Performance: predicted vs actual %change in closing price', figsize=(15, 10))
   ```

Store Models

You can save and load `sklearn` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the file path where the model will be stored.

3. Call the `dump` method with the model and file path.

   ```
   joblib.dump(model, file_name)
   ```

   If you dump the model using the `joblib` module before you save the model, you don't need to retrain the model.

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method with the model key.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call `GetFilePath` with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the path where the model is stored.

3. Call `load` with the file path.

   ```
   loaded_model = joblib.load(file_name)
   ```

   This method returns the saved model.

7.5 Hmmlearn

Introduction

This page explains how to build, train, test, and store `Hmmlearn` models.

Import Libraries

Import the `Hmmlearn` library.

```
from hmmlearn import hmm
import joblib
```

You need the `joblib` library to store models.

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. Follow these steps to prepare the data:

1. Select the close column of the historical data DataFrame.

   ```
   closes = history['close']
   ```

2. Call the `pct_change` method and then drop the first row.

   ```
   daily_returns = closes.pct_change().iloc[1:]
   ```

3. Call the `reshape` method.

   ```
   X = daily_returns.values.reshape(-1, 1)
   ```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, assume the market has only 2 regimes and the market returns follow a Gaussian distribution. Therefore, create a 2-component Hidden Markov Model with Gaussian emissions, which is equivalent to a Gaussian mixture model with 2 means. Follow these steps to create the model:

1. Call the `GaussianHMM` constructor with the number of components, a covariance type, and the number of iterations.

   ```
   model = hmm.GaussianHMM(n_components=2, covariance_type="full", n_iter=100)
   ```

2. Call the `fit` method with the training data.

   ```
   model.fit(X)
   ```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Call the predict method with the testing dataset.

   ```
   y = model.predict(X)
   ```

2. Plot the regimes in a scatter plot.

   ```
   plt.figure(figsize=(15, 10))
   plt.scatter(ret.index, [f'Regime {n+1}' for n in y])
   plt.title(f'{symbol} market regime')
   plt.xlabel("time")
   plt.show()
   ```

Store Models

You can save and load `Hmmlearn` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the file path where the model will be stored.

3. Call the `dump` method with the model and file path.

   ```
   joblib.dump(model, file_name)
   ```

   If you dump the model using the `joblib` module before you save the model, you don't need to retrain the model.

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the path where the model is stored.

3. Call the `load` method with the file path.

   ```
   loaded_model = joblib.load(file_name)
   ```

   This method returns the saved model.

7.6 Gplearn

Introduction

This page introduces how to build, train, test, and store `GPlearn` models.

Import Libraries

Import the `GPlearn` library.

```
from gplearn.genetic import SymbolicRegressor, SymbolicTransformer
from sklearn.model_selection import train_test_split
import joblib
```

You need the `sklearn` library to prepare the data and the `joblib` library to store models.

Get Historical Data

[Get some historical market data](#) to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some [historical data](#) to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	Daily percent change of the open, high, low, close, and volume of the SPY over the last 5 days
Labels	Daily percent return of the SPY over the next day

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Call the `pct_change` method and then drop the first row.

   ```
   daily_returns = history['close'].pct_change()[1:]
   ```

2. Loop through the `daily_returns` DataFrame and collect the features and labels.

   ```
   n_steps = 5
   features = []
   labels = []
   for i in range(len(daily_returns)-n_steps):
       features.append(daily_returns.iloc[i:i+n_steps].values)
       labels.append(daily_returns.iloc[i+n_steps])
   ```

3. Convert the lists of features and labels into `numpy` arrays.

   ```
   X = np.array(features)
   y = np.array(labels)
   ```

4. Split the data into training and testing periods.

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, y)
   ```

Train Models

You need to [prepare the historical data](#) for training before you train the model. If you have prepared the data, build and train the model. In this example, create a Symbolic Transformer to generate new non-linear features and then build a Symbolic Regressor model. Follow these steps to create the model:

1. Declare a set of functions to use for feature engineering.

   ```
   function_set = ['add', 'sub', 'mul', 'div',
                   'sqrt', 'log', 'abs', 'neg', 'inv',
                   'max', 'min']
   ```

2. Call the `SymbolicTransformer` constructor with the preceding set of functions.

   ```
   gp_transformer = SymbolicTransformer(function_set=function_set,
                                        random_state=0,
                                        verbose=1)
   ```

3. Call the `fit` method with the training features and labels.

   ```
   gp_transformer.fit(X_train, y_train)
   ```

   This method displays the following output:

4. Call the `transform` method with the original features.

   ```
   gp_features_train = gp_transformer.transform(X_train)
   ```

5. Call the `hstack` method with the original features and the transformed features.

   ```
   new_X_train = np.hstack((X_train, gp_features_train))
   ```

6. Call the `SymbolicRegressor` constructor.

   ```
   gp_regressor = SymbolicRegressor(random_state=0, verbose=1)
   ```

7. Call the `fit` method with the engineered features and the original labels.

   ```
   gp_regressor.fit(new_X_train, y_train)
   ```

Test Models

You need to [build and train the model](#) before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Feature engineer the testing set data.

   ```
   gp_features_test = gp_transformer.transform(X_test)
   new_X_test = np.hstack((X_test, gp_features_test))
   ```

2. Call the `predict` method with the engineered testing set data.

   ```
   y_predict = gp_regressor.predict(new_X_test)
   ```

3. Plot the actual and predicted labels of the testing period.

   ```
   df = pd.DataFrame({'Real': y_test.flatten(), 'Predicted': y_predict.flatten()})
   df.plot(title='Model Performance: predicted vs actual closing price', figsize=(15, 10))
   plt.show()
   ```

4. Calculate the R-square value.

```
r2 = gp_regressor.score(new_X_test, y_test)
print(f"The explained variance of the GP model: {r2*100:.2f}%")
```

Store Models

You can save and load `GPlearn` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key names of the models to be stored in the ObjectStore.

```
transformer_key = "transformer"
regressor_key = "regressor"
```

2. Call the `GetFilePath` method with the key names.

```
transformer_file = qb.ObjectStore.GetFilePath(transformer_key)
regressor_file = qb.ObjectStore.GetFilePath(regressor_key)
```

This method returns the file paths where the models will be stored.

3. Call the `dump` method with the models and file paths.

```
joblib.dump(gp_transformer, transformer_file)
joblib.dump(gp_regressor, regressor_file)
```

If you dump the model using the `joblib` module before you save the model, you don't need to retrain the model.

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method.

```
qb.ObjectStore.ContainsKey(transformer_key)
qb.ObjectStore.ContainsKey(regressor_key)
```

This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the keys.

```
transformer_file = qb.ObjectStore.GetFilePath(transformer_key)
regressor_file = qb.ObjectStore.GetFilePath(regressor_key)
```

This method returns the path where the model is stored.

3. Call the `load` method with the file paths.

```
loaded_transformer = joblib.load(transformer_file)
loaded_regressor = joblib.load(regressor_file)
```

This method returns the saved models.

7.7 PyTorch

Introduction

This page explains how how to build, train, test, and store `PyTorch` models.

Import Libraries

Import the `torch` , `sklearn` , and `joblib` libraries by the following:

```
import torch
from torch import nn
from sklearn.model_selection import train_test_split
import joblib
```

You need the `sklearn` library to prepare the data and the `joblib` library to store models.

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	The last 5 closing prices
Labels	The following day's closing price

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Perform fractional differencing on the historical data.

   ```
   df = (history['close'] * 0.5 + history['close'].diff() * 0.5)[1:]
   ```

   Fractional differencing helps make the data stationary yet retains the variance information.

2. Loop through the `df` DataFrame and collect the features and labels.

   ```
   n_steps = 5
   features = []
   labels = []
   for i in range(len(df)-n_steps):
       features.append(df.iloc[i:i+n_steps].values)
       labels.append(df.iloc[i+n_steps])
   ```

3. Convert the lists of features and labels into `numpy` arrays.

   ```
   features = np.array(features)
   labels = np.array(labels)
   ```

4. Standardize the features and labels

   ```
   X = (features - features.mean()) / features.std()
   y = (labels - labels.mean()) / labels.std()
   ```

5. Split the data into training and testing periods.

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, y)
   ```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, create a deep neural network with 2 hidden layers. Follow these steps to create the model:

1. Define a subclass of `nn.Module` to be the model.

   In this example, use the ReLU activation function for each layer.

   ```
   class NeuralNetwork(nn.Module):
       # Model Structure
       def __init__(self):
           super(NeuralNetwork, self).__init__()
           self.flatten = nn.Flatten()
           self.linear_relu_stack = nn.Sequential(
               nn.Linear(5, 5),   # input size, output size of the layer
               nn.ReLU(),         # Relu non-linear transformation
               nn.Linear(5, 5),
               nn.ReLU(),
               nn.Linear(5, 1),   # Output size = 1 for regression
           )

       # Feed-forward training/prediction
       def forward(self, x):
           x = torch.from_numpy(x).float()   # Convert to tensor in type float
           result = self.linear_relu_stack(x)
           return result
   ```

2. Create an instance of the model and set its configuration to train on the GPU if it's available.

   ```
   device = 'cuda' if torch.cuda.is_available() else 'cpu'
   model = NeuralNetwork().to(device)
   ```

3. Set the loss and optimization functions.

   In this example, use the mean squared error as the loss function and stochastic gradient descent as the optimizer.

   ```
   loss_fn = nn.MSELoss()
   learning_rate = 0.001
   optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
   ```

4. Train the model.

   In this example, train the model through 5 epochs.

   ```
   epochs = 5
   for t in range(epochs):
       print(f"Epoch {t+1}\n-------------------------------")

       # Since we're using SGD, we'll be using the size of data as batch number.
       for batch, (X, y) in enumerate(zip(X_train, y_train)):
           # Compute prediction and loss
           pred = model(X)
   ```

```
        real = torch.from_numpy(np.array(y).flatten()).float()
        loss = loss_fn(pred, real)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch
            print(f"loss: {loss:.5f}  [{current:5d}/{len(X_train):5d}]")
```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Predict with the testing data.

   ```
   predict = model(X_test)
   y_predict = predict.detach().numpy()   # Convert tensor to numpy ndarray
   ```

2. Plot the actual and predicted values of the testing period.

   ```
   df = pd.DataFrame({'Real': y_test.flatten(), 'Predicted': y_predict.flatten()})
   df.plot(title='Model Performance: predicted vs actual standardized fractional return', figsize=(15, 10))
   plt.show()
   ```

3. Calculate the R-square value.

   ```
   r2 = 1 - np.sum(np.square(y_test.flatten() - y_predict.flatten())) / np.sum(np.square(y_test.flatten() - y_test.mean()))
   print(f"The explained variance by the model (r-square): {r2*100:.2f}%")
   ```

Store Models

You can save and load `PyTorch` models using the ObjectStore.

Save Models

Don't use the `torch.save` method to save models because the tensor data will be lost and corrupt the save. Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the file path where the model will be stored.

3. Call the `dump` method with the model and file path.

   ```
   joblib.dump(model, file_name)
   ```

   If you dump the model using the `joblib` module before you save the model, you don't need to retrain the model.

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key`, save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the path where the model is stored.

3. Call the `load` method with the file path.

   ```
   loaded_model = joblib.load(file_name)
   ```

   This method returns the saved model.

7.8 Stable Baselines

Introduction

This page introduces how to use `stable baselines` library in Python for reinforcement machine learning (RL) model building, training, saving in the ObjectStore, and loading, through an example of a single-asset deep Q-network learning (DQN) trading bot.

Import Libraries

Import the `stable_baselines` , and `gym` .

```
import gym
from stable_baselines import DQN
from stable_baselines.deepq.policies import MlpPolicy
```

Get Historical Data

[Get some historical market data]to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, calculate the log return time-series of the securities:

```
ret = np.log(history/history.shift(1)).iloc[1:].close
```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the environment and the model. In this example, create a `gym` environment to initialize the training environment, agent and reward. Then, create a RL model by DQN algorithm. Follow these steps to create the environment and the model:

1. Split the data for training and testing to evaluate our model.

   ```
   X_train = history.iloc[:-50].values
   X_test = history.iloc[-50:].values
   y_train = ret.iloc[:-50].values
   y_test = ret.iloc[-50:].values
   ```

2. Create a custom `gym` environment class.

   In this example, create a custom environment with previous 5 OHLCV log-return data as observation and the highest portfolio value as reward.

   ```
   class TradingEnv(gym.Env):
       metadata = {'render.modes': ['console']}

       FLAT = 0
       LONG = 1
       SHORT = 2

       def __init__(self, ohlcv, ret):
           super(TradingEnv, self).__init__()

           self.ohlcv = ohlcv
           self.ret = ret
           self.trading_cost = 0.01
           self.reward = 1

           # The number of step the training has taken, starts at 5 since we're using the previous 5 data for observation.
           self.current_step = 5
           # The last action
           self.last_action = 0

           # Define action and observation space
           # Example when using discrete actions, we have 3: LONG, SHORT and FLAT.
           n_actions = 3
           self.action_space = gym.spaces.Discrete(n_actions)
           # The observation will be the coordinate of the agent, shape for (5 previous data poionts, OHLCV)
           self.observation_space = gym.spaces.Box(low=-np.inf, high=np.inf, shape=(5, 5), dtype=np.float64)

       def reset(self):
           # Reset the number of step the training has taken
           self.current_step = 5
           # Reset the last action
           self.last_action = 0
           # must return np.array type
           return self.ohlcv[self.current_step-5:self.current_step].astype(np.float32)

       def step(self, action):
           if action == self.LONG:
               self.reward *= 1 + self.ret[self.current_step] - (self.trading_cost if self.last_action != action else 0)
           elif action == self.SHORT:
               self.reward *= 1 + -1 * self.ret[self.current_step] - (self.trading_cost if self.last_action != action else 0)
           elif action == self.FLAT:
                self.reward *= 1 - (self.trading_cost if self.last_action != action else 0)
           else:
               raise ValueError("Received invalid action={} which is not part of the action space".format(action))

           self.last_action = action
           self.current_step += 1

           # Have we iterate all data points?
           done = (self.current_step == self.ret.shape[0]-1)

           # Reward as return
           return self.ohlcv[self.current_step-5:self.current_step].astype(np.float32), self.reward, done, {}

       def render(self, mode='console'):
           if mode != 'console':
               raise NotImplementedError()
           print(f'Equity Value: {self.reward}')
   ```

3. Initialize the environment.

   ```
   env = TradingEnv(X_train, y_train)
   ```

4. Train the model.

   In this example, create a RL model and train with MLP-policy DQN algorithm.

   ```
   model = DQN(MlpPolicy, env, verbose=1)
   model.learn(total_timesteps=1000)
   ```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Initialize a list to store the equity value with initial capital in each timestep, and variables to store last action and trading cost.

   ```
   equity = [1]
   last_action = 0
   ```

```
trading_cost = 0.01
```

2. Iterate each testing data point for prediction and trading.

```
for i in range(5, X_test.shape[0]):
    action, _ = model.predict(X_test[i-5:i], deterministic=True)

    if action == 0:
        new = equity[-1] * (1 - (trading_cost if last_action != action else 0))
    elif action == 1:
        new = equity[-1] * (1 + y_test[i] - (trading_cost if last_action != action else 0))
    elif action == 2:
        new = equity[-1] * (1 + -1 * y_test[i] - (trading_cost if last_action != action else 0))

    equity.append(new)
    last_action = action
```

3. Plot the result.

```
plt.figure(figsize=(15, 10))
plt.title("Equity Curve")
plt.xlabel("timestep")
plt.ylabel("equity")
plt.plot(equity)
plt.show()
```

Store Models

You can save and load `stable baselines` models using the ObjectStore.

Save Models

1. Set the key name of the model to be stored in the ObjectStore.

```
model_key = "model"
```

2. Call the `GetFilePath` method with the key.

```
file_name = qb.ObjectStore.GetFilePath(model_key)
```

This method returns the file path where the model will be stored.

3. Call the `save` method with the file path.

```
model.save(file_name)
```

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method.

```
qb.ObjectStore.ContainsKey(model_key)
```

This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the key.

```
file_name = qb.ObjectStore.GetFilePath(model_key)
```

This method returns the path where the model is stored.

3. Call the `load` method with the file path, environment and policy.

```
loaded_model = DQN.load(file_name, env=env, policy=MlpPolicy)
```

This method returns the saved model.

7.9 Tslearn

Introduction

This page explains how to build, train, test, and store `tslearn` models.

Import Libraries

Import the `tslearn` libraries.

```
from tslearn.barycenters import softdtw_barycenter
from tslearn.clustering import TimeSeriesKMeans
```

Get Historical Data

Get some historical market data to train and test the model. For example, get data for the securities shown in the following table:

Group Name	Tickers
Overall US market	SPY, QQQ, DIA
Tech companies	AAPL, MSFT, TSLA
Long-term US Treasury ETFs	IEF, TLT
Short-term US Treasury ETFs	SHV, SHY
Heavy metal ETFs	GLD, IAU, SLV
Energy sector	USO, XLE, XOM

```
qb = QuantBook()
tickers = ["SPY", "QQQ", "DIA",
           "AAPL", "MSFT", "TSLA",
           "IEF", "TLT", "SHV", "SHY",
           "GLD", "IAU", "SLV",
           "USO", "XLE", "XOM"]
symbols = [qb.AddEquity(ticker, Resolution.Daily).Symbol for ticker in tickers]
history = qb.History(symbols, datetime(2020, 1, 1), datetime(2022, 2, 20))
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, standardize the log close price time-series of the securities. Follow these steps to prepare the data:

1. Unstack the historical DataFrame and select the close column.

   ```
   close = history.unstack(0).close
   ```

2. Take the logarithm of the historical time series.

   ```
   log_close = np.log(close)
   ```

   Taking the logarithm eases the compounding effect.

3. Standardize the data.

   ```
   standard_close = (log_close - log_close.mean()) / log_close.std()
   ```

Train Models

Instead of using real-time comparison, we could apply a technique call Dynamic Time Wrapping (DTW) with Barycenter Averaging (DBA). Intuitively, it is a technique of averaging a few time-series into a single one without losing much of their information. Since not all time-series would move efficiently like in ideal EMH assumption, this would allow similarity analysis of different time-series with sticky lags. Check the technical details from tslearn documentation page .

We then can separate different clusters by KMean after DBA.

```
# Set up the Time Series KMean model with soft DBA.
km = TimeSeriesKMeans(n_clusters=6,   # We have 6 main groups
                      metric="softdtw",  # soft for differentiable
                      random_state=0)

# Fit the model.
km.fit(standard_close.T)
```

Test Models

We visualize the clusters and their corresponding underlying series.

1. Predict with the label of the data.

   ```
   labels = km.predict(standard_close.T)
   ```

2. Create a class to aid plotting.

   ```
   def plot_helper(ts):
       # plot all points of the data set
       for i in range(ts.shape[0]):
           plt.plot(ts[i, :], "k-", alpha=.2)

       # plot the given barycenter of them
       barycenter = softdtw_barycenter(ts, gamma=1.)
       plt.plot(barycenter, "r-", linewidth=2)
   ```

3. Plot the results.

   ```
   j = 1
   plt.figure(figsize=(15, 10))
   for i in set(labels):
       # Select the series in the i-th cluster.
       X = standard_close.iloc[:, [n for n, k in enumerate(labels) if k == i]].values

       # Plot the series and barycenter-averaged series.
       plt.subplot(len(set(labels)) // 3 + (1 if len(set(labels))%3 != 0 else 0), 3, j)
       plt.title(f"Cluster {i+1}")
       plot_helper(X.T)

       j += 1

   plt.show()
   ```

4. Display the groupings.

   ```
   for i in set(labels):
       print(f"Cluster {i+1}: {standard_close.columns[[n for n, k in enumerate(labels) if k == i]]}")
   ```

Store Models

You can save and load `tslearn` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the file path where the model will be stored.

3. Call the `to_hdf5` method with the file path.

   ```
   km.to_hdf5(file_name + ".hdf5")
   ```

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key`, save the model using the `model_key` before you proceed.

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the path where the model is stored.

3. Call the `from_hdf5` method with the file path.

   ```
   loaded_model = TimeSeriesKMeans.from_hdf5(file_name + ".hdf5")
   ```

   This method returns the saved model.

Reference

- F. Petitjean, A. Ketterlin, P. Gancarski. (2010). A global averaging method for dynamic time warping, with applications to clustering. *Pattern Recognition. 44(2011). 678-693. Retreived from https://lig-membres.imag.fr/bisson/cours/M2INFO-AIW-ML/papers/PetitJean11.pdf*

7.10 XGBoost

Introduction

This page explains how to build, train, test, and store `XGBoost` models.

Import Libraries

Import the `xgboost` , `sklearn` , and `joblib` libraries.

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
import joblib
```

You need the `sklearn` library to prepare the data and the `joblib` library to save models.

Get Historical Data

[Get some historical market data]to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some [historical data]to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	The last 5 closing prices
Labels	The following day's closing price

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Perform fractional differencing on the historical data.

   ```
   df = (history['close'] * 0.5 + history['close'].diff() * 0.5)[1:]
   ```

   Fractional differencing helps make the data stationary yet retains the variance information.

2. Loop through the `df` DataFrame and collect the features and labels.

   ```
   n_steps = 5
   features = []
   labels = []
   for i in range(len(df)-n_steps):
       features.append(df.iloc[i:i+n_steps].values)
       labels.append(df.iloc[i+n_steps])
   ```

3. Convert the lists of features and labels into `numpy` arrays.

   ```
   features = np.array(features)
   labels = np.array(labels)
   ```

4. Standardize the features and labels

   ```
   X = (features - features.mean()) / features.std()
   y = (labels - labels.mean()) / labels.std()
   ```

5. Split the data into training and testing periods.

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, y)
   ```

Train Models

We're about to train a gradient-boosted random forest for future price prediction.

1. Split the data for training and testing to evaluate our model.

   ```
   X_train, X_test, y_train, y_test = train_test_split(X, y)
   ```

2. Format training set into XGBoost matrix.

   ```
   dtrain = xgb.DMatrix(X_train, label=y_train)
   ```

3. Train the model with parameters.

   ```
   params = {
     'booster': 'gbtree',
     'colsample_bynode': 0.8,
     'learning_rate': 0.1,
     'lambda': 0.1,
     'max_depth': 5,
     'num_parallel_tree': 100,
     'objective': 'reg:squarederror',
     'subsample': 0.8,
   }
   model = xgb.train(params, dtrain, num_boost_round=10)
   ```

Test Models

We then make predictions on the testing data set. We compare our Predicted Values with the Expected Values by plotting both to see if our Model has predictive power.

1. Format testing set into XGBoost matrix.

   ```
   dtest = xgb.DMatrix(X_test, label=y_test)
   ```

2. Predict with the testing set data.

   ```
   y_predict = model.predict(dtest)
   ```

3. Plot the result.

   ```
   df = pd.DataFrame({'Real': y_test.flatten(), 'Predicted': y_predict.flatten()})
   df.plot(title='Model Performance: predicted vs actual closing price', figsize=(15, 10))
   plt.show()
   ```

Store Models

Saving the Model

We dump the model using the `joblib` module and save it to `ObjectStore` file path. This way, the model doesn't need to be retrained, saving time and computational resources.

1. Set the key name of the model to be stored in the ObjectStore.

```
model_key = "model"
```

2. Call `GetFilePath` with the key's name to get the file path.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

3. Call dump with the model and file path to save the model to the file path.

   ```
   joblib.dump(model, file_name)
   ```

Loading the Model

Let's retrieve the model from `ObjectStore` file path and load by `joblib` .

1. Call the `ContainsKey` method.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call `GetFilePath` with the key's name to get the file path.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

3. Call `load` with the file path to fetch the saved model.

   ```
   loaded_model = joblib.load(file_name)
   ```

To ensure loading the model was successfuly, let's test the model.

```
y_pred = loaded_model.predict(dtest)
df = pd.DataFrame({'Real': y_test.flatten(), 'Predicted': y_pred.flatten()})
df.plot(title='Model Performance: predicted vs actual closing price', figsize=(15, 10))
```

7.11 Aesera

Introduction

This page explains how to build, train, test, and store `Aesera` models.

Import Libraries

Import the `aesera` , and `sklearn` libraries.

```
import aesara
import aesara.tensor as at
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import joblib
```

You need the `joblib` library to store models.

Get Historical Data

Get some historical market data to train and test the model. For example, to get data for the SPY ETF during 2020 and 2021, run:

```
qb = QuantBook()
symbol = qb.AddEquity("SPY", Resolution.Daily).Symbol
history = qb.History(symbol, datetime(2020, 1, 1), datetime(2022, 1, 1)).loc[symbol]
```

Prepare Data

You need some historical data to prepare the data for the model. If you have historical data, manipulate it to train and test the model. In this example, use the following features and labels:

Data Category	Description
Features	Normalized close price of the SPY over the last 5 days
Labels	Return direction of the SPY over the next day

The following image shows the time difference between the features and labels:

Follow these steps to prepare the data:

1. Obtain the close price and return direction series.

```
close = history['close']
returns = data['close'].pct_change().shift(-1)[lookback*2-1:-1].reset_index(drop=True)
labels = pd.Series([1 if y > 0 else 0 for y in returns])   # binary class
```

2. Loop through the `close` Series and collect the features.

```
lookback = 5
lookback_series = []
for i in range(1, lookback + 1):
    df = data['close'].shift(i)[lookback:-1]
    df.name = f"close-{i}"
    lookback_series.append(df)
X = pd.concat(lookback_series, axis=1)
# Normalize using the 5 day interval
X = MinMaxScaler().fit_transform(X.T).T[4:]
```

3. Convert the lists of features and labels into `numpy` arrays.

```
X = np.array(features)
y = np.array(labels)
```

4. Split the data into training and testing periods.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Train Models

You need to prepare the historical data for training before you train the model. If you have prepared the data, build and train the model. In this example, build a Logistic Regression model with log loss cross entropy and square error as cost function. Follow these steps to create the model:

1. Generate a dataset.

```
# D = (input_values, target_class)
D = (np.array(X_train), np.array(y_train))
```

2. Initialize variables.

```
# Declare Aesara symbolic variables
x = at.dmatrix("x")
y = at.dvector("y")

# initialize the weight vector w randomly using share so model coefficients keep their values
# between training iterations (updates)
rng = np.random.default_rng(100)
w = aesara.shared(rng.standard_normal(X.shape[1]), name="w")

# initialize the bias term
b = aesara.shared(0., name="b")
```

3. Construct the model graph.

```
# Construct Aesara expression graph
p_1 = 1 / (1 + at.exp(-at.dot(x, w) - b))       # Logistic transformation
prediction = p_1 > 0.5                          # The prediction thresholded
xent = y * at.log(p_1) - (1 - y) * at.log(1 - p_1)  # Cross-entropy log-loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()      # The cost to minimize (MSE)
gw, gb = at.grad(cost, [w, b])                  # Compute the gradient of the cost
```

4. Compile the model.

```
train = aesara.function(
          inputs=[x, y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = aesara.function(inputs=[x], outputs=prediction)
```

5. Train the model with training dataset.

```
pred, err = train(D[0], D[1])

# We can also inspect the final outcome
print("Final model:")
print(w.get_value())
print(b.get_value())
print("target values for D:")
print(D[1])
print("prediction on D:")
print(predict(D[0]))    # whether > 0.5 or not
```

Test Models

You need to build and train the model before you test its performance. If you have trained the model, test it on the out-of-sample data. Follow these steps to test the model:

1. Call the `predict` method with the features of the testing period.

   ```
   y_hat = predict(np.array(X_test))
   ```

2. Plot the actual and predicted labels of the testing period.

   ```
   df = pd.DataFrame({'y': y_test, 'y_hat': y_hat}).astype(int)
   df.plot(title='Model Performance: predicted vs actual return direction in closing price', figsize=(12, 5))
   ```

3. Calculate the prediction accuracy.

   ```
   correct = sum([1 if x==y else 0 for x, y in zip(y_test, y_hat)])
   print(f"Accuracy: {correct}/{y_test.shape[0]} ({correct/y_test.shape[0]}%)")
   ```

Store Models

You can save and load `aesera` models using the ObjectStore.

Save Models

Follow these steps to save models in the ObjectStore:

1. Set the key name of the model to be stored in the ObjectStore.

   ```
   model_key = "model"
   ```

2. Call the `GetFilePath` method with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the file path where the model will be stored.

3. Call the `dump` method with the model and file path.

   ```
   joblib.dump(predict, file_name)
   ```

   If you dump the model using the `joblib` module before you save the model, you don't need to retrain the model.

Load Models

You must save a model into the ObjectStore before you can load it from the ObjectStore. If you saved a model, follow these steps to load it:

1. Call the `ContainsKey` method with the model key.

   ```
   qb.ObjectStore.ContainsKey(model_key)
   ```

   This method returns a boolean that represents if the `model_key` is in the ObjectStore. If the ObjectStore does not contain the `model_key` , save the model using the `model_key` before you proceed.

2. Call `GetFilePath` with the key.

   ```
   file_name = qb.ObjectStore.GetFilePath(model_key)
   ```

   This method returns the path where the model is stored.

3. Call `load` with the file path.

   ```
   loaded_model = joblib.load(file_name)
   ```

   This method returns the saved model.

8 Debugging

Introduction

The debugger is a built-in tool to help you debug coding errors while in the Research Environment. The debugger enables you to slow down the code execution, step through the program line-by-line, and inspect the variables to understand the internal state of the notebook.

The Research Environment debugger isn't currently available for C#.

Breakpoints

Breakpoints are lines in your notebook where execution pauses. You need at least one breakpoint in your notebook to start the debugger. [Open a project](#) to start adjusting its breakpoints.

Add Breakpoints

Click to the left of a line to add a breakpoint on that line.

Edit Breakpoint Conditions

Follow these steps to customize what happens when a breakpoint is hit:

1. Right-click the breakpoint and then click Edit Breakpoint... .
2. Click one of the options in the following table:

Option	Additional Steps	Description
Expression	Enter an expression and then press Enter .	The breakpoint only pauses the notebook when the expression is true.
Hit Count	Enter an integer and then press Enter .	The breakpoint doesn't pause the notebook until its hit the number of times you specify.

Enable and Disable Breakpoints

To enable a breakpoint, right-click it and then click Enable Breakpoint .

To disable a breakpoint, right-click it and then click Disable Breakpoint .

Follow these steps to enable and disable all breakpoints:

1. In the right navigation menu, click the Run and Debug icon.
2. In the Run and Debug panel, hover over the Breakpoints section and then click the Toggle Active Breakpoints icon.

Remove Breakpoints

To remove a breakpoint, right-click it and then click Remove Breakpoint .

Follow these steps to remove all breakpoints:

1. In the right navigation menu, click the Run and Debug icon.
2. In the Run and Debug panel, hover over the Breakpoints section and then click the Remove All Breakpoints icon.

Launch Debugger

Follow these steps to launch the debugger:

1. [Open the project](#) you want to debug.
2. [Open the notebook file](#) in your project.
3. In a notebook cell, add at least one breakpoint.
4. In the top-left corner of the cell, click the drop-down arrow and then click Debug Cell .

If the Run and Debug panel is not open, it opens when the first breakpoint is hit.

Control Debugger

After you launch the debugger, you can use the following buttons to control it:

Button	Name	Default Keyboard Shortcut	Description
	Continue		Continue execution until the next breakpoint
	Step Over	Alt+F10	Step to the next line of code in the current or parent scope
	Step Into	Alt+F11	Step into the definition of the function call on the current line
	Restart	Shift+F11	Restart the debugger
	Disconnect	Shift+F5	Exit the debugger

Inspect Variables

After you launch the debugger, you can inspect the state of your notebook as it executes each line of code. You can inspect local variables or custom expressions. The values of variables in your notebook are formatted in the IDE to improve readability. For example, if you inspect a variable that references a DataFrame, the debugger represents the variable value as the following:

Local Variables

The Variables section of the Run and Debug panel shows the local variables at the current breakpoint. If a variable in the panel is an object, click it to see its members. The panel updates as the notebook runs.

Follow these steps to update the value of a variable:

1. In the Run and Debug panel, right-click a variable and then click Set Value .
2. Enter the new value and then press Enter .

Custom Expressions

The Watch section of the Run and Debug panel shows any custom expressions you add. For example, you can add an expression to show a `datetime` object.

Follow these steps to add a custom expression:

1. Hover over the Watch section and then click the plus icon that appears.
2. Enter an expression and then press Enter .

9 Meta Analysis

9.1 Key Concepts

Introduction

Understanding your strategy trades in detail is key to attributing performance, and determining areas to focus for improvement. This analysis can be done with the QuantConnect API. We enable you to load backtest, optimization, and live trading results into the Research Environment.

Backtest Analysis

Load your backtest results into the Research Environment to analyze trades and easily compare them against the raw backtesting data. For more information on loading and manipulating backtest results, see Backtest Analysis .

Optimization Analysis

Load your optimization results into the Research Environment to analyze how different combinations of parameters affect the algorithm's performance. For more information on loading and manipulating optimizations results, see Optimization Analysis .

Live Analysis

Load your live trading results into the Research Environment to compare live trading performance against simulated backtest results, or analyze your trades to improve your slippage and fee models. For more information on loading and manipulating live trading results, see Live Analysis .

9.2 Backtest Analysis

Introduction

Load your backtest results into the Research Environment to analyze trades and easily compare them against the raw backtesting data. Compare backtests from different projects to find uncorrelated strategies to combine for better performance.

Loading your backtest trades allows you to plot fills against detailed data, or locate the source of profits. Similarly you can search for periods of high churn to reduce turnover and trading fees.

Read Backtest Results

To get the results of a backtest, call the `ReadBacktest` method with the project Id and backtest ID.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Api;

var backtest = api.ReadBacktest(projectId, backtestId);
```

```
backtest = api.ReadBacktest(project_id, backtest_id)
```

To get the project Id, open the project in the Algorithm Lab and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

To get the backtest Id, open a backtest result in the Algorithm Lab and check the last line of its log file . An example backtest Id is 97e7717f387cadd070e4b77015aacece.

Note that this method returns a snapshot of the backtest at the current moment. If the backtest is still executing, the result won't include all of the backtest data.

The `ReadBacktest` method returns a `Backtest` object, which have the following attributes:

Plot Order Fills

Follow these steps to plot the daily order fills of a backtest:

1. Get the backtest orders.

   ```
   orders = api.ReadBacktestOrders(project_id, backtest_id)
   ```

   To get the project Id, open the project in the Algorithm Lab and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

   To get the backtest Id, open a backtest result in the Algorithm Lab and check the last line of its log file . An example backtest Id is 97e7717f387cadd070e4b77015aacece.

   The `ReadBacktestOrders` method returns a list of `Order` objects, which have the following properties:

2. Organize the trade times and prices for each security into a dictionary.

   ```
   class OrderData:
       def __init__(self):
           self.buy_fill_times = []
           self.buy_fill_prices = []
           self.sell_fill_times = []
           self.sell_fill_prices = []

   order_data_by_symbol = {}
   for order in orders:
       if order.Symbol not in order_data_by_symbol:
           order_data_by_symbol[order.Symbol] = OrderData()
       order_data = order_data_by_symbol[order.Symbol]
       is_buy = order.Quantity > 0
       (order_data.buy_fill_times if is_buy else order_data.sell_fill_times).append(order.LastFillTime.date())
       (order_data.buy_fill_prices if is_buy else order_data.sell_fill_prices).append(order.Price)
   ```

3. Get the price history of each security you traded.

   ```
   qb = QuantBook()
   start_date = datetime.max.date()
   end_date = datetime.min.date()
   for symbol, order_data in order_data_by_symbol.items():
       start_date = min(start_date, min(order_data.buy_fill_times), min(order_data.sell_fill_times))
       end_date = max(end_date, max(order_data.buy_fill_times), max(order_data.sell_fill_times))
   start_date -= timedelta(days=1)
   all_history = qb.History(list(order_data_by_symbol.keys()), start_date, end_date, Resolution.Daily)
   ```

4. Create a candlestick plot for each security and annotate each plot with buy and sell markers.

   ```
   import plotly.express as px
   import plotly.graph_objects as go

   for symbol, order_data in order_data_by_symbol.items():
       history = all_history.loc[symbol]

       # Plot security price candlesticks
       candlestick = go.Candlestick(x=history.index,
                                    open=history['open'],
                                    high=history['high'],
                                    low=history['low'],
                                    close=history['close'],
                                    name='Price')
       layout = go.Layout(title=go.layout.Title(text=f'{symbol.Value} Trades'),
                          xaxis_title='Date',
                          yaxis_title='Price',
                          xaxis_rangeslider_visible=False,
                          height=600)
       fig = go.Figure(data=[candlestick], layout=layout)

       # Plot buys
       fig.add_trace(go.Scatter(
           x=order_data.buy_fill_times,
           y=order_data.buy_fill_prices,
           marker=go.scatter.Marker(color='aqua', symbol='triangle-up', size=10),
           mode='markers',
           name='Buys',
       ))

       # Plot sells
       fig.add_trace(go.Scatter(
           x=order_data.sell_fill_times,
           y=order_data.sell_fill_prices,
           marker=go.scatter.Marker(color='indigo', symbol='triangle-down', size=10),
           mode='markers',
           name='Sells',
       ))

   fig.show()
   ```

   Note: The preceding plots only show the last fill of each trade. If your trade has partial fills, the plots only display the last fill.

Plot Metadata

Follow these steps to plot the equity curve, benchmark, and drawdown of a backtest:

1. Get the backtest instance.

```
backtest = api.ReadBacktest(project_id, backtest_id)
```

   To get the project Id, [open the project in the Algorithm Lab](#) and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

   To get the backtest Id, [open a backtest result in the Algorithm Lab](#) and check the last line of its [log file](#). An example backtest Id is 97e7717f387cadd070e4b77015aacece.

2. Get the "Strategy Equity", "Drawdown", and "Benchmark" `Chart` objects.

```
equity_chart = backtest.Charts["Strategy Equity"]
drawdown_chart = backtest.Charts["Drawdown"]
benchmark_chart = backtest.Charts["Benchmark"]
```

3. Get the "Equity", "Equity Drawdown", and "Benchmark" `Series` from the preceding charts.

```
equity = equity_chart.Series["Equity"].Values
drawdown = drawdown_chart.Series["Equity Drawdown"].Values
benchmark = benchmark_chart.Series["Benchmark"].Values
```

4. Create a `pandas.DataFrame` from the series values.

```
df = pd.DataFrame({
    "Equity": pd.Series({datetime.fromtimestamp(value.x): value.y for value in equity}),
    "Drawdown": pd.Series({datetime.fromtimestamp(value.x): value.y for value in drawdown}),
    "Benchmark": pd.Series({datetime.fromtimestamp(value.x): value.y for value in benchmark})
}).ffill()
```

5. Plot the performance chart.

```
# Create subplots to plot series on same/different plots
fig, ax = plt.subplots(2, 1, figsize=(12, 12), sharex=True, gridspec_kw={'height_ratios': [2, 1]})

# Plot the equity curve
ax[0].plot(df.index, df["Equity"])
ax[0].set_title("Strategy Equity Curve")
ax[0].set_ylabel("Portfolio Value ($)")

# Plot the benchmark on the same plot, scale by using another y-axis
ax2 = ax[0].twinx()
ax2.plot(df.index, df["Benchmark"], color="grey")
ax2.set_ylabel("Benchmark Price ($)", color="grey")

# Plot the drawdown on another plot
ax[1].plot(df.index, df["Drawdown"], color="red")
ax[1].set_title("Drawdown")
ax[1].set_xlabel("Time")
ax[1].set_ylabel("%")
```

The following table shows all the chart series you can plot:

Chart	Series	Description
Strategy Equity	Equity	Time series of the equity curve
	Daily Performance	Time series of daily percentage change
Capacity	Strategy Capacity	Time series of [strategy capacity](#) snapshots
Drawdown	Equity Drawdown	Time series of equity peak-to-trough value
Benchmark	Benchmark	Time series of the [benchmark](#) closing price (SPY, by default)
Exposure	SecurityType - Long Ratio	Time series of the overall ratio of SecurityType long positions of the whole portfolio if any SecurityType is ever in the universe
	SecurityType - Short Ratio	Time series of the overall ratio of SecurityType short position of the whole portfolio if any SecurityType is ever in the universe
Custom Chart	Custom Series	Time series of a `Series` in a [custom chart](#)

9.3 Optimization Analysis

Introduction

Load your optimization results into the Research Environment to analyze how different combinations of parameters affect the algorithm's performance.

Read Optimization Results

To get the results of an optimization, call the `ReadOptimization` method with the optimization Id.

```
var optimization = api.ReadOptimization(optimizationId);
```

```
optimization = api.ReadOptimization(optimization_id)
```

To get the optimization Id, check the [Cloud Terminal] when you [run an optimization in the Algorithm Lab] . An example optimization Id is O-696d861d6dbbed45a8442659bd24e59f.

The `ReadOptimization` method returns an `Optimization` object, which have the following attributes:

9.4 Live Analysis

Introduction

Load your live trading results into the Research Environment to compare live trading performance against simulated backtest results.

Read Live Results

To get the results of a live algorithm, call the `ReadLiveAlgorithm` method with the project Id and deployment ID.

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Api;

var liveAlgorithm = api.ReadLiveAlgorithm(projectId, deployId);
```

```
live_algorithm = api.ReadLiveAlgorithm(project_id, deploy_id)
```

To get the project Id, open the project in the Algorithm Lab and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

To get the deployment Id, open a live result in the Algorithm Lab and check its log file . An example deployment Id is L-ac54ffadf4ca52efabcd1ac29e4735cf. If you have deployed the project multiple times, the log file has multiple deployment Ids. In this case, use the most recent Id.

The `ReadLiveAlgorithm` method returns a `LiveAlgorithmResults` object, which have the following attributes:

Reconciliation

Reconciliation is a way to quantify the difference between an algorithm's live performance and its out-of-sample (OOS) performance (a backtest run over the live deployment period).

Seeing the difference between live performance and OOS performance gives you a way to determine if the algorithm is making unrealistic assumptions, exploiting data differences, or merely exhibiting behavior that is impractical or impossible in live trading.

A perfectly reconciled algorithm has an exact overlap between its live equity and OOS backtest curves. Any deviation means that the performance of the algorithm has differed for some reason. Several factors can contribute to this, often stemming from the algorithm design.

Live Deployment Reconciliation

Reconciliation is scored using two metrics: returns correlation and dynamic time warping (DTW) distance.

What is DTW Distance?

Dynamic Time Warp (DTW) Distance quantifies the difference between two time-series. It is an algorithm that measures the shortest path between the points of two time-series. It uses Euclidean distance as a measurement of **point-to-point distance** and returns an overall measurement of the distance on the scale of the initial time-series values. We apply DTW to the returns curve of the live and OOS performance, so the DTW distance measurement is on the scale of percent returns.

$$\begin{equation} DTW(X,Y) = min\bigg\{\sum_{l=1}^{L}\left(x_{m_l} - y_{n_l}\right)^{2}\in P^{N\times M}\bigg\} \end{equation}$$

For the reasons outlined in our research notebook on the topic (linked below), QuantConnect annualizes the daily DTW. An annualized distance provides a user with a measurement of the annual difference in the magnitude of returns between the two curves. A perfect score is 0, meaning the returns for each day were precisely the same. A DTW score of 0 is nearly impossible to achieve, and we consider anything below 0.2 to be a decent score. A distance of 0.2 means the returns between an algorithm's live and OOS performance deviated by 20% over a year.

What is Returns Correlation?

Returns correlation is the simple Pearson correlation between the live and OOS returns. Correlation gives us a rudimentary understanding of how the returns move together. Do they trend up and down at the same time? Do they deviate in direction or timing?

$$\begin{equation} \rho_{XY} = \frac{cov(X, Y)}{\sigma_X\sigma_Y} \end{equation}$$

An algorithm's returns correlation should be as close to 1 as possible. We consider a good score to be 0.8 or above, meaning that there is a strong positive correlation. This indicates that the returns move together most of the time and that for any given return you see from one of the curves, the other curve usually has a similar direction return (positive or negative).

Why Do We Need Both DTW and Returns Correlation?

Each measurement provides insight into distinct elements of time-series similarity, but neither measurement alone gives us the whole picture. Returns correlation tells us whether or not the live and OOS returns move together, but it doesn't account for the possible differences in the magnitude of the returns. DTW distance measures the difference in magnitude of returns but provides no insight into whether or not the returns move in the same direction. It is possible for there to be two cases of equity curve similarity where both pairs have the same DTW distance, but one has perfectly negatively correlated returns, and the other has a perfectly positive correlation. Similarly, it is possible for two pairs of equity curves to each have perfect correlation but substantially different DTW distance. Having both measurements provides us with a more comprehensive understanding of the actual similarity between live and OOS performance. We outline several interesting cases and go into more depth on the topic of reconciliation in research we have published.

Plot Order Fills

Follow these steps to plot the daily order fills of a live algorithm:

- Get the live trading orders.

  ```
  orders = api.ReadLiveOrders(project_id)
  ```

  To get the project Id, open the project in the Algorithm Lab and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

  By default, the orders with an ID between 0 and 100. To get orders with an ID greater than 100, pass `start` and `end` arguments to the `ReadLiveOrders` method. Note that `end - start` must be less than 100.

  ```
  orders = api.ReadLiveOrders(project_id, 100, 150)
  ```

  The `ReadLiveOrders` method returns a list of `Order` objects, which have the following properties:

- Organize the trade times and prices for each security into a dictionary.

  ```
  class OrderData:
      def __init__(self):
          self.buy_fill_times = []
          self.buy_fill_prices = []
          self.sell_fill_times = []
          self.sell_fill_prices = []

  order_data_by_symbol = {}
  for order in orders:
      if order.Symbol not in order_data_by_symbol:
          order_data_by_symbol[order.Symbol] = OrderData()
      order_data = order_data_by_symbol[order.Symbol]
      is_buy = order.Quantity > 0
      (order_data.buy_fill_times if is_buy else order_data.sell_fill_times).append(order.LastFillTime.date())
      (order_data.buy_fill_prices if is_buy else order_data.sell_fill_prices).append(order.Price)
  ```

- Get the price history of each security you traded.

  ```
  qb = QuantBook()
  start_date = datetime.max.date()
  end_date = datetime.min.date()
  for symbol, order_data in order_data_by_symbol.items():
      start_date = min(start_date, min(order_data.buy_fill_times), min(order_data.sell_fill_times))
      end_date = max(end_date, max(order_data.buy_fill_times), max(order_data.sell_fill_times))
  start_date -= timedelta(days=1)
  all_history = qb.History(list(order_data_by_symbol.keys()), start_date, end_date, Resolution.Daily)
  ```

- Create a candlestick plot for each security and annotate each plot with buy and sell markers.

```
import plotly.express as px
import plotly.graph_objects as go

for symbol, order_data in order_data_by_symbol.items():
    history = all_history.loc[symbol]

    # Plot security price candlesticks
    candlestick = go.Candlestick(x=history.index,
                                open=history['open'],
                                high=history['high'],
                                low=history['low'],
                                close=history['close'],
                                name='Price')
    layout = go.Layout(title=go.layout.Title(text=f'{symbol.Value} Trades'),
                       xaxis_title='Date',
                       yaxis_title='Price',
                       xaxis_rangeslider_visible=False,
                       height=600)
    fig = go.Figure(data=[candlestick], layout=layout)

    # Plot buys
    fig.add_trace(go.Scatter(
        x=order_data.buy_fill_times,
        y=order_data.buy_fill_prices,
        marker=go.scatter.Marker(color='aqua', symbol='triangle-up', size=10),
        mode='markers',
        name='Buys',
    ))

    # Plot sells
    fig.add_trace(go.Scatter(
        x=order_data.sell_fill_times,
        y=order_data.sell_fill_prices,
        marker=go.scatter.Marker(color='indigo', symbol='triangle-down', size=10),
        mode='markers',
        name='Sells',
    ))

fig.show()
```

Note: The preceding plots only show the last fill of each trade. If your trade has partial fills, the plots only display the last fill.

Plot Metadata

Follow these steps to plot the equity curve, benchmark, and drawdown of a live algorithm:

1. Get the live algorithm instance.

```
live_algorithm = api.ReadLiveAlgorithm(project_id, deploy_id)
```

To get the project Id, open the project in the Algorithm Lab and check the URL. For example, the project Id of https://www.quantconnect.com/project/13946911 is 13946911.

To get the deployment Id, open a live result in the Algorithm Lab and check its log file . An example deployment Id is L-ac54ffadf4ca52efabcd1ac29e4735cf. If you have deployed the project multiple times, the log file has multiple deployment Ids. In this case, use the most recent Id.

2. Get the results of the live algorithm.

```
results = live_algorithm.LiveResults.Results
```

3. Get the "Strategy Equity", "Drawdown", and "Benchmark" `Chart` objects.

```
equity_chart = results.Charts["Strategy Equity"]
drawdown_chart = results.Charts["Drawdown"]
benchmark_chart = results.Charts["Benchmark"]
```

4. Get the "Equity", "Equity Drawdown", and "Benchmark" `Series` from the preceding charts.

```
equity = equity_chart.Series["Equity"].Values
drawdown = drawdown_chart.Series["Equity Drawdown"].Values
benchmark = benchmark_chart.Series["Benchmark"].Values
```

5. Create a `pandas.DataFrame` from the series values.

```
df = pd.DataFrame({
    "Equity": pd.Series({datetime.fromtimestamp(value.x): value.y for value in equity}),
    "Drawdown": pd.Series({datetime.fromtimestamp(value.x): value.y for value in drawdown}),
    "Benchmark": pd.Series({datetime.fromtimestamp(value.x): value.y for value in benchmark})
}).ffill()
```

6. Plot the performance chart.

```
# Create subplots to plot series on same/different plots
fig, ax = plt.subplots(2, 1, figsize=(12, 12), sharex=True, gridspec_kw={'height_ratios': [2, 1]})

# Plot the equity curve
ax[0].plot(df.index, df["Equity"])
ax[0].set_title("Strategy Equity Curve")
ax[0].set_ylabel("Portfolio Value ($)")

# Plot the benchmark on the same plot, scale by using another y-axis
ax2 = ax[0].twinx()
ax2.plot(df.index, df["Benchmark"], color="grey")
ax2.set_ylabel("Benchmark Price ($)", color="grey")

# Plot the drawdown on another plot
ax[1].plot(df.index, df["Drawdown"], color="red")
ax[1].set_title("Drawdown")
ax[1].set_xlabel("Time")
ax[1].set_ylabel("%")
```

The following table shows all the chart series you can plot:

Chart	Series	Description
Strategy Equity	Equity	Time series of the equity curve
	Daily Performance	Time series of daily percentage change
Capacity	Strategy Capacity	Time series of strategy capacity snapshots
Drawdown	Equity Drawdown	Time series of equity peak-to-trough value
Benchmark	Benchmark	Time series of the benchmark closing price (SPY, by default)
Exposure	SecurityType - Long Ratio	Time series of the overall ratio of SecurityType long positions of the whole portfolio if any SecurityType is ever in the universe
	SecurityType - Short Ratio	Time series of the overall ratio of SecurityType short position of the whole portfolio if any SecurityType is ever in the universe
Custom Chart	Custom Series	Time series of a `Series` in a custom chart

10 Applying Research

10.1 Key Concepts

Introduction

The ultimate goal of research is to produce a strategy that you can backtest and eventually trade live. Once you've developed a hypothesis that you're confident in, you can start working towards exporting your research into backtesting. To export the code, you need to replace `QuantBook()` with `self` and replace the [QuantBook methods](#) with their `QCAlgorithm` counterparts.

Workflow

Imagine that you've developed the following hypothesis: stocks that are below 1 standard deviation of their 30-day mean are due to revert and increase in value. The following Research Environment code picks out such stocks from a preselected basket of stocks:

```
import numpy as np
qb = QuantBook()

symbols = {}
assets = ["SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
"SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
"VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]

for i in range(len(assets)):
    symbols[assets[i]] = qb.AddEquity(assets[i],Resolution.Minute).Symbol

# Fetch history on our universe
df = qb.History(qb.Securities.Keys, 30, Resolution.Daily)

# Make all of them into a single time index.
df = df.close.unstack(level=0)

# Calculate the truth value of the most recent price being less than 1 std away from the mean
classifier = df.le(df.mean().subtract(df.std())).tail(1)

# Get indexes of the True values
classifier_indexes = np.where(classifier)[1]

# Get the Symbols for the True values
classifier = classifier.transpose().iloc[classifier_indexes].index.values

# Get the std values for the True values (used for magnitude)
magnitude = df.std().transpose()[classifier_indexes].values

# Zip together to iterate over later
selected = zip(classifier, magnitude)
```

Once you are confident in your hypothesis, you can export this code into the backtesting environment. The algorithm will ultimately go long on the stocks that pass the classifier logic. One way to accommodate this model into a backtest is to create a [Scheduled Event](#) that uses the model to pick stocks and place orders.

```
def Initialize(self) -> None:
    self.SetStartDate(2014, 1, 1)
    self.SetCash(1000000)
    self.SetBenchmark("SPY")

    self.SetPortfolioConstruction(EqualWeightingPortfolioConstructionModel())
    self.SetExecution(ImmediateExecutionModel())

    self.assets = ["IEF", "SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
                   "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                   "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]

    self.symbols = {}

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.symbols[self.assets[i]] = self.AddEquity(self.assets[i], Resolution.Minute).Symbol

    # Set the Scheduled Event method
    self.Schedule.On(self.DateRules.Every(DayOfWeek.Monday), self.TimeRules.AfterMarketOpen("IEF", 1), self.EveryDayAfterMarketOpen)
```

Now that the `Initialize` method of the algorithm is set, export the model into the Scheduled Event method. You just need to switch `qb` with `self` and replace `QuantBook` methods with their `QCAlgorithm` counterparts. In this example, you don't need to switch any methods because the model only uses methods that exist in `QCAlgorithm` .

```
def EveryDayAfterMarketOpen(self):
    qb = self
    # Fetch history on our universe
    df = qb.History(qb.Securities.Keys, 5, Resolution.Daily)

    # Make all of them into a single time index.
    df = df.close.unstack(level=0)

    # Calculate the truth value of the most recent price being less than 1 std away from the mean
    classifier = df.le(df.mean().subtract(df.std())).tail(1)

    # Get indexes of the True values
    classifier_indexes = np.where(classifier)[1]

    # Get the Symbols for the True values
    classifier = classifier.transpose().iloc[classifier_indexes].index.values

    # Get the std values for the True values (used for magnitude)
    magnitude = df.std().transpose()[classifier_indexes].values

    # Zip together to iterate over later
    selected = zip(classifier, magnitude)

    # ==============================

    insights = []

    for symbol, magnitude in selected:
        insights.append(Insight.Price(symbol, timedelta(days=5), InsightDirection.Up, magnitude))

    self.EmitInsights(insights)
```

With the Research Environment model now in the backtesting environment, you can further analyze its performance with its [backtesting metrics](#) . If you are confident in the backtest, you can eventually live trade this strategy.

To view full examples of this Research to Production workflow, see the examples in the menu.

Contribute Tutorials

If you contribute Research to Production tutorials, you'll get the following benefits:

- A [QCC](#) reward
- You'll learn the Research to Production methodology to improve your own strategy research and development
- Your contribution will be featured in the community forum

To view the topics the community wants Research to Production tutorials for, see [the issues with the WishList tag in the Research GitHub repository](#) . If you find a topic you want to create a tutorial for, make a pull request to the repository with your tutorial and we will review it.

To request new tutorial topics, [contact us](#) .

10.2 Mean Reversion

Introduction

This page explains how to you can use the Research Environment to develop and test a Mean Reversion hypothesis, then put the hypothesis in production.

Create Hypothesis

Imagine that we've developed the following hypothesis: stocks that are below 1 standard deviation of their 30-day-mean are due to revert and increase in value, statistically around 85% chance if we assume the return series is stationary and the price series is a Random Process. We've developed the following code in research to pick out such stocks from a preselected basket of stocks.

Import Libraries

Load the required assembly files and data types.

We'll need to import libraries to help with data processing. Import `numpy` and `scipy` libraries by the following:

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Research;
using System;
using MathNet.Numerics.Distributions;
```

```
import numpy as np
from scipy.stats import norm, zscore
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   var qb = new QuantBook();
   ```

   ```
   qb = QuantBook()
   ```

2. Select the desired tickers for research.

   ```
   var assets = new List<string>() {"SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
                     "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                     "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"};
   ```

   ```
   assets = ["SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
             "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
             "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]
   ```

3. Call the `AddEquity` method with the tickers, and their corresponding resolution.

   ```
   foreach(var ticker in assets){
       qb.AddEquity(ticker, Resolution.Minute);
   }
   ```

   ```
   for i in range(len(assets)):
       qb.AddEquity(assets[i],Resolution.Minute)
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   var history = qb.History(qb.Securities.Keys, new DateTime(2021, 1, 1), new DateTime(2021, 12, 31), Resolution.Daily);
   ```

   ```
   history = qb.History(qb.Securities.Keys, datetime(2021, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data to get an extent of the signal on how much the stock is deviated from its norm for each ticker.

1. Extract close prices for each `Symbol` from `Slice` data.
2. Select the close column and then call the `unstack` method.

   ```
   var closes = new Dictionary<Symbol, List<Decimal>>();
   foreach(var slice in history){
       foreach(var symbol in slice.Keys){
           if(!closes.ContainsKey(symbol)){
               closes.Add(symbol, new List<Decimal>());
           }
           closes[symbol].Add(slice.Bars[symbol].Close);
       }
   }
   ```

   ```
   df = history['close'].unstack(level=0)
   ```

3. Get the 30-day rolling mean, standard deviation series, z-score and filtration for each `Symbol` .
4. Calculate the truth value of the most recent price being less than 1 standard deviation away from the mean price.

   ```
   var rollingMean = new Dictionary<Symbol, List<double>>();
   var rollingStd = new Dictionary<Symbol, List<double>>();
   var filter = new Dictionary<Symbol, List<bool>>();
   var zScore = new Dictionary<Symbol, List<double>>();
   foreach(var kvp in closes)
   {
       var symbol = kvp.Key;
       if(!rollingMean.ContainsKey(symbol)){
           rollingMean.Add(symbol, new List<double>());
           rollingStd.Add(symbol, new List<double>());
           zScore.Add(symbol, new List<double>());
           filter.Add(symbol, new List<bool>());
       }
       for (int i=30; i < closes.Values.ElementAt(0).Count; i++)
       {
           var slice = kvp.Value.Skip(i).Take(30);
           rollingMean[symbol].Add(decimal.ToDouble(slice.Average()));
           rollingStd[symbol].Add(Math.Sqrt(slice.Average(v => Math.Pow(decimal.ToDouble(v-slice.Average()), 2))));
           zScore[symbol].Add((decimal.ToDouble(closes[symbol][i]) - rollingMean[symbol].Last()) / rollingStd[symbol].Last());
           filter[symbol].Add(zScore[symbol].Last() < -1);
       }
   }
   ```

   ```
   classifier = df.le(df.rolling(30).mean() - df.rolling(30).std())
   ```

5. Calculate the expected return and its probability, then calculate the weight.
6. Get the z-score for the True values, then compute the expected return and probability (used for Insight magnitude and confidence).

   ```
   var magnitude = new Dictionary<Symbol, List<double>>();
   var confidence = new Dictionary<Symbol, List<double>>();
   var weights = new Dictionary<Symbol, List<double>>();
   foreach(var kvp in rollingMean)
   {
   ```

```
        var symbol = kvp.Key;
        if(!magnitude.ContainsKey(symbol)){
            magnitude.Add(symbol, new List<double>());
            confidence.Add(symbol, new List<double>());
            weights.Add(symbol, new List<double>());
        }
        for (int i=1; i < rollingMean.Values.ElementAt(0).Count; i++)
        {
            magnitude[symbol].Add(-zScore[symbol][i] * rollingStd[symbol][i] / decimal.ToDouble(closes[symbol][i-1]));
            confidence[symbol].Add(Normal.CDF(0, 1, -zScore[symbol][i]));
            // Filter if trade or not
            var trade = filter[symbol][i] ? 1d : 0d;
            weights[symbol].Add(trade * Math.Max(confidence[symbol].Last() - 1 / (magnitude[symbol].Last() + 1), 0));
        }
    }
```

```
z_score = df.apply(zscore)[classifier]
magnitude = -z_score * df.rolling(30).std() / df.shift(1)
confidence = (-z_score).apply(norm.cdf)
```

7. Convert the weights into 2-d array.
8. Call `fillna` to fill NaNs with 0.

```
double[,] weight = new double[weights.Values.ElementAt(0).Count, weights.Count];
int j = 0;
foreach(var symbol in weights.Keys){
    for(int i=0; i < weights[symbol].Count; i++){
        weight[i, j] = weights[symbol][i];
    }
    j++;
}
```

```
magnitude.fillna(0, inplace=True)
confidence.fillna(0, inplace=True)
```

9. Get our trading weight, we'd take a long only portfolio and normalized to total weight = 1.

```
public double[,] Normalize(double[,] array)
{
    for(int i=0; i < array.GetLength(0); i++)
    {
        var sum = 0.0;
        for (int j=0; j < array.GetLength(1); j++)
        {
            sum += array[i, j];
        }
        if (sum == 0.0) continue;
        for (int j=0; j < array.GetLength(1); j++)
        {
            array[i, j] = array[i, j] / sum;
        }
    }

    return array;
}
weight = Normalize(weight);
```

```
weight = confidence - 1 / (magnitude + 1)
weight = weight[weight > 0].fillna(0)
sum_ = np.sum(weight, axis=1)
for i in range(weight.shape[0]):
    if sum_[i] > 0:
        weight.iloc[i] = weight.iloc[i] / sum_[i]
    else:
        weight.iloc[i] = 0
weight = weight.iloc[:-1]
```

Test Hypothesis

We would test the performance of this strategy. To do so, we would make use of the calculated weight for portfolio optimization.

1. Convert close price to 2-d array.

```
double[,] close = new double[closes.Values.ElementAt(0).Count, closes.Count];
int j = 0;
foreach(var symbol in closes.Keys){
    for(int i=0; i < closes[symbol].Count; i++){
        close[i, j] = decimal.ToDouble(closes[symbol][i]);
    }
    j++;
}
```

2. Get the total daily return series.

```
var totalValue = new List<double>{1.0};
var dailySum = 0.0;
for(int i=0; i < weight.GetLength(0) - 1; i++)
{
    totalValue.Add(totalValue.Last() * (1 + dailySum));
    dailySum = 0.0;
    for (int j=0; j < weight.GetLength(1); j++)
    {
        if (close[i, j] != 0 && double.IsFinite(close[i+1, j]) && double.IsFinite(close[i, j]) && double.IsFinite(weight[i, j]))
        {
            dailySum += weight[i, j] * (close[i+1, j] - close[i, j]) / close[i, j];
        }
    }
}
```

```
ret = pd.Series(index=range(df.shape[0] - 1))
for i in range(df.shape[0] - 1):
    ret[i] = weight.iloc[i] @ df.pct_change().iloc[i + 1].T
```

3. Call `cumprod` to get the cumulative return.

```
total_ret = (ret + 1).cumprod()
```

4. Set index for visualization.

```
total_ret.index = weight.index
```

5. Display the result.

```
for(int i=0; i < totalValue.Count; i=i+5)
{
    Console.WriteLine("Portfolio Value in Day{0}: {1}", i, totalValue[i]);
}
```

```
total_ret.plot(title='Strategy Equity Curve', figsize=(15, 10))
plt.show()
```

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into research is to create a scheduled event which uses our model to pick stocks and goes long.

```
private List<string> _asset = new List<string>{"SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
        "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
        "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"};
```

```
public override void Initialize()
{
    // 1. Required: Five years of backtest history
    SetStartDate(2014, 1, 1);

    // 2. Required: Alpha Streams Models:
    SetBrokerageModel(BrokerageName.AlphaStreams);

    // 3. Required: Significant AUM Capacity
    SetCash(1000000);

    // 4. Required: Benchmark to SPY
    SetBenchmark("SPY");

    SetPortfolioConstruction(new InsightWeightingPortfolioConstructionModel());
    SetExecution(new ImmediateExecutionModel());

    // Add Equity ------------------------------------------------
    foreach(var ticker in _asset)
    {
    AddEquity(ticker, Resolution.Minute);
    }

    // Set Scheduled Event Method For Our Model
    Schedule.On(DateRules.EveryDay(),
        TimeRules.BeforeMarketClose("SHY", 5),
        EveryDayBeforeMarketClose);
}
```

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2014, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.SetPortfolioConstruction(InsightWeightingPortfolioConstructionModel())
    self.SetExecution(ImmediateExecutionModel())

    self.assets = ["SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
                   "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                   "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.AddEquity(self.assets[i], Resolution.Minute)

    # Set Scheduled Event Method For Our Model
    self.Schedule.On(self.DateRules.EveryDay(), self.TimeRules.BeforeMarketClose("SHY", 5), self.EveryDayBeforeMarketClose)
```

Now we export our model into the scheduled event method. We will remove `qb` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
private void EveryDayBeforeMarketClose()
{
    // Fetch history on our universe
    var history = History(Securities.Keys, 30, Resolution.Daily);
    if (history.Count() < 0) return;

    // Extract close prices for each Symbol from Slice data
    var closes = new Dictionary<Symbol, List<Decimal>>();
    foreach(var slice in history){
        foreach(var symbol in slice.Keys){
            if(!closes.ContainsKey(symbol)){
                closes.Add(symbol, new List<Decimal>());
            }
            closes[symbol].Add(slice.Bars[symbol].Close);
        }
    }

    // Get the 30-day rolling mean, standard deviation series, z-score and filtration for each Symbol
    var rollingMean = new Dictionary<string, double>();
    var rollingStd = new Dictionary<string, double>();
    var filter = new Dictionary<string, bool>();
    var zScore = new Dictionary<string, double>();
    foreach(var kvp in closes)
    {
        var symbol = kvp.Key;
        if(!rollingMean.ContainsKey(symbol)){
            rollingMean.Add(symbol, decimal.ToDouble(kvp.Value.Average()));
            rollingStd.Add(symbol, Math.Sqrt(kvp.Value.Average(v => Math.Pow(decimal.ToDouble(v-kvp.Value.Average()), 2))));
            zScore.Add(symbol, (decimal.ToDouble(kvp.Value.Last()) - rollingMean[symbol]) / rollingStd[symbol]);
            filter.Add(symbol, zScore[symbol] < -1);
        }
    }

    // Calculate the expected return and its probability, then calculate the weight
    var magnitude = new Dictionary<Symbol, double>();
    var confidence = new Dictionary<Symbol, double>();
    var weights = new Dictionary<Symbol, double>();
    foreach(var kvp in rollingMean)
    {
        var symbol = kvp.Key;
        if(!magnitude.ContainsKey(symbol)){
            magnitude.Add(symbol, -zScore[symbol] * rollingStd[symbol] / decimal.ToDouble(closes[symbol].Last()));
            confidence.Add(symbol, Normal.CDF(0, 1, -zScore[symbol]));
            // Filter if trade or not
            var trade = filter[symbol] ? 1d : 0d;
            weights.Add(symbol, trade * Math.Max(confidence[symbol] - 1 / (magnitude[symbol] + 1), 0));
        }
    }

    // Normalize the weights, then emit insights
    var sum = weights.Sum(x => x.Value);
    if (sum == 0) return;

    foreach(var kvp in weights)
    {
        var symbol = kvp.Key;
        weights[symbol] = kvp.Value / sum;

        var insight = new Insight(symbol, TimeSpan.FromDays(1), InsightType.Price, InsightDirection.Up, magnitude[symbol], confidence[symbol], null, weights[symbol]);
        EmitInsights(insight);
    }
}
```

```
def EveryDayBeforeMarketClose(self) -> None:
    qb = self
    # Fetch history on our universe
```

```
    df = qb.History(qb.Securities.Keys, 30, Resolution.Daily)
    if df.empty: return

    # Make all of them into a single time index.
    df = df.close.unstack(level=0)

    # Calculate the truth value of the most recent price being less than 1 std away from the mean
    classifier = df.le(df.mean().subtract(df.std())).iloc[-1]
    if not classifier.any(): return

    # Get the z-score for the True values, then compute the expected return and probability
    z_score = df.apply(zscore)[[classifier.index[i] for i in range(classifier.size) if classifier.iloc[i]]]

    magnitude = -z_score * df.std() / df
    confidence = (-z_score).apply(norm.cdf)

    # Get the latest values
    magnitude = magnitude.iloc[-1].fillna(0)
    confidence = confidence.iloc[-1].fillna(0)

    # Get the weights, then zip together to iterate over later
    weight = confidence - 1 / (magnitude + 1)
    weight = weight[weight > 0].fillna(0)
    sum_ = np.sum(weight)
    if sum_ > 0:
        weight = (weight) / sum_
        selected = zip(weight.index, magnitude, confidence, weight)
    else:
        return

    # ==============================

    insights = []

    for symbol, magnitude, confidence, weight in selected:
        insights.append( Insight.Price(symbol, timedelta(days=1), InsightDirection.Up, magnitude, confidence, None, weight) )

    self.EmitInsights(insights)
```

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics

Total Trades	5835
Average Win	0.34%
Average Loss	-0.20%
Compounding Annual Return	-10.596%
Drawdown	75.500%
Expectancy	-0.140
Net Profit	-63.869%
Sharpe Ratio	-0.269
Probabilistic Sharpe Ratio	0.000%
Loss Rate	68%
Win Rate	32%
Profit-Loss Ratio	1.67
Alpha	-0.079
Beta	0.275
Annual Standard Deviation	0.204
Annual Variance	0.042
Information Ratio	-0.626
Tracking Error	0.227
Treynor Ratio	-0.199
Total Fees	$78061.92
Estimated Strategy Capacity	$9300000.00
Lowest Capacity Asset	TMV UBTUG7D0B7TX

```
#region imports
from AlgorithmImports import *
from scipy.stats import norm, zscore
#endregion

class MeanReversionDemo(QCAlgorithm):

    def Initialize(self):
        #1. Required: Five years of backtest history
        self.SetStartDate(2014, 1, 1)
```

- Charts
- Statistics
- Code
  - Main.cs
  - Research.ipynb

Clone Algorithm

Overall Statistics	
Total Trades	13445
Average Win	0.27%
Average Loss	-0.07%
Compounding Annual Return	-10.332%
Drawdown	74.600%
Expectancy	-0.166
Net Profit	-62.888%
Sharpe Ratio	-0.26
Probabilistic Sharpe Ratio	0.000%
Loss Rate	82%
Win Rate	18%
Profit-Loss Ratio	3.57
Alpha	-0.077
Beta	0.274
Annual Standard Deviation	0.204
Annual Variance	0.042
Information Ratio	-0.618
Tracking Error	0.227
Treynor Ratio	-0.193
Total Fees	$86686.70
Estimated Strategy Capacity	$180000.00
Lowest Capacity Asset	TMV UBTUG7D0B7TX

```
/// Please refer to Research.ipynb
using QuantConnect.Algorithm.Framework.Alphas;
using QuantConnect.Algorithm.Framework.Portfolio;
using QuantConnect.Algorithm.Framework.Execution;
using QuantConnect.Brokerages;
using System;
using System.Collections.Generic;
using System.Linq;
using MathNet.Numerics.Distributions;
```

10.3 Random Forest Regression

Introduction

This page explains how to you can use the Research Environment to develop and test a Random Forest Regression hypothesis, then put the hypothesis in production.

Create Hypothesis

We've assumed the price data is a time series with some auto regressive property (i.e. its expectation is related to past price information). Therefore, by using past information, we could predict the next price level. One way to do so is by Random Forest Regression, which is a supervised machine learning algorithm where its weight and bias is decided in non-linear hyperdimension.

Import Libraries

We'll need to import libraries to help with data processing and machine learning. Import `sklearn` , `numpy` and `matplotlib` libraries by the following:

```
from sklearn.ensemble import RandomForestRegressor
import numpy as np
from matplotlib import pyplot as plt
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the desired tickers for research.

   ```
   symbols = {}
   assets = ["SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
             "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
             "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]
   ```

3. Call the `AddEquity` method with the tickers, and their corresponding resolution. Then store their `Symbol` s.

   ```
   for i in range(len(assets)):
       symbols[assets[i]] = qb.AddEquity(assets[i],Resolution.Minute).Symbol
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   history = qb.History(qb.Securities.Keys, datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data as well as to build the ML model before testing the hypothesis. Our methodology is to use fractional differencing close price as the input data in order to (1) provide stationarity, and (2) retain sufficient extent of variance of the previous price information. We assume d=0.5 is the right balance to do so.

1. Select the close column and then call the `unstack` method.

   ```
   df = history['close'].unstack(level=0)
   ```

2. Feature engineer the data as fractional differencing for input.

   ```
   input_ = df.diff() * 0.5 + df * 0.5
   input_ = input_.iloc[1:]
   ```

3. Shift the data for 1-step backward as training output result.

   ```
   output = df.shift(-1).iloc[:-1]
   ```

4. Split the data into training and testing sets.

   ```
   splitter = int(input_.shape[0] * 0.8)
   X_train = input_.iloc[:splitter]
   X_test = input_.iloc[splitter:]
   y_train = output.iloc[:splitter]
   y_test = output.iloc[splitter:]
   ```

5. Initialize a Random Forest Regressor.

   ```
   regressor = RandomForestRegressor(n_estimators=100, min_samples_split=5, random_state = 1990)
   ```

6. Fit the regressor.

   ```
   regressor.fit(X_train, y_train)
   ```

Test Hypothesis

We would test the performance of this ML model to see if it could predict 1-step forward price precisely. To do so, we would compare the predicted and actual prices.

1. Predict the testing set.

   ```
   predictions = regressor.predict(X_test)
   ```

2. Convert result into `DataFrame` .

   ```
   predictions = pd.DataFrame(predictions, index=y_test.index, columns=y_test.columns)
   ```

3. Plot the result for comparison.

   ```
   for col in y_test.columns:
       plt.figure(figsize=(15, 10))

       y_test[col].plot(label="Actual")
       predictions[col].plot(label="Prediction")

       plt.title(f"{col} Regression Result")
       plt.legend()
       plt.show()
       plt.clf()
   ```

   For more plots, please [clone the project](#) and run the notebook.

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into backtest is to create a scheduled event which uses our model to predict the expected return. Since we could calculate the expected return, we'd use Mean-Variance Optimization for portfolio construction.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2014, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)
```

```
    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.SetPortfolioConstruction(MeanVarianceOptimizationPortfolioConstructionModel(portfolioBias = PortfolioBias.Long,
                                                                                      period=252))
    self.SetExecution(ImmediateExecutionModel())

    self.assets = ["SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
                   "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                   "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.AddEquity(self.assets[i], Resolution.Minute)

    # Initialize the timer to train the Machine Learning model
    self.time = datetime.min

    # Set Scheduled Event Method For Our Model
    self.Schedule.On(self.DateRules.EveryDay(), self.TimeRules.BeforeMarketClose("SHY", 5), self.EveryDayBeforeMarketClose)
```

We'll also need to create a function to train and update our model from time to time.

```
def BuildModel(self) -> None:
    # Initialize the Random Forest Regressor
    self.regressor = RandomForestRegressor(n_estimators=100, min_samples_split=5, random_state = 1990)

    # Get historical data
    history = self.History(self.Securities.Keys, 360, Resolution.Daily)

    # Select the close column and then call the unstack method.
    df = history['close'].unstack(level=0)

    # Feature engineer the data for input.
    input_ = df.diff() * 0.5 + df * 0.5
    input_ = input_.iloc[1:].ffill().fillna(0)

    # Shift the data for 1-step backward as training output result.
    output = df.shift(-1).iloc[:-1].ffill().fillna(0)

    # Fit the regressor
    self.regressor.fit(input_, output)
```

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
def EveryDayBeforeMarketClose(self) -> None:
    # Retrain the regressor every month
    if self.time < self.Time:
        self.BuildModel()
        self.time = Expiry.EndOfMonth(self.Time)

    qb = self
    # Fetch history on our universe
    df = qb.History(qb.Securities.Keys, 2, Resolution.Daily)
    if df.empty: return

    # Make all of them into a single time index.
    df = df.close.unstack(level=0)

    # Feature engineer the data for input
    input_ = df.diff() * 0.5 + df * 0.5
    input_ = input_.iloc[-1].fillna(0).values.reshape(1, -1)

    # Predict the expected price
    predictions = self.regressor.predict(input_)

    # Get the expected return
    predictions = (predictions - df.iloc[-1].values) / df.iloc[-1].values
    predictions = predictions.flatten()

    # ==============================

    insights = []

    for i in range(len(predictions)):
        insights.append( Insight.Price(self.assets[i], timedelta(days=1), InsightDirection.Up, predictions[i]) )

    self.EmitInsights(insights)
```

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	45282
Average Win	0.01%
Average Loss	0.00%
Compounding Annual Return	0.013%
Drawdown	12.700%
Expectancy	-0.011
Net Profit	0.118%
Sharpe Ratio	0.015
Probabilistic Sharpe Ratio	0.018%
Loss Rate	69%
Win Rate	31%
Profit-Loss Ratio	2.16
Alpha	0.003
Beta	-0.035
Annual Standard Deviation	0.021
Annual Variance	0
Information Ratio	-0.559
Tracking Error	0.155
Treynor Ratio	-0.009
Total Fees	$48708.25
Estimated Strategy Capacity	$1100000.00
Lowest Capacity Asset	BIL TT1EBZ21QWKL

```
#region imports
from AlgorithmImports import *
from sklearn.ensemble import RandomForestRegressor
#endregion

class RandomForestRegressionDemo(QCAlgorithm):

    def Initialize(self):
        #1. Required: Five years of backtest history
        self.SetStartDate(2014, 1, 1)
```

10.4 Uncorrelated Assets

Introduction

This page explains how to you can use the Research Environment to develop and test a Uncorrelated Assets hypothesis, then put the hypothesis in production.

Create Hypothesis

According to Modern Portfolio Thoery, asset combinations with negative or very low correlation could have lower total portfolio variance given the same level of return. Thus, uncorrelated assets allows you to find a portfolio that will, theoretically, be more diversified and resilient to extreme market events. We're testing this statement in real life scenario, while hypothesizing a portfolio with uncorrelated assets could be a consistent portfolio. In this example, we'll compare the performance of 5-least-correlated-asset portfolio (proposed) and 5-most-correlated-asset portfolio (benchmark), both equal weighting.

Import Libraries

Load the required assembly files and data types.

We'll need to import libraries to help with data processing and visualization. Import `numpy` and `matplotlib` libraries by the following:

```
#load "../Initialize.csx"
#load "../QuantConnect.csx"

using QuantConnect;
using QuantConnect.Data;
using QuantConnect.Data.Market;
using QuantConnect.Algorithm;
using QuantConnect.Research;
using System;
using System.Linq;
using Accord.Statistics;
```

```
import numpy as np
from matplotlib import pyplot as plt
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook`.

```
var qb = new QuantBook();
```

```
qb = QuantBook()
```

2. Select the desired tickers for research.

```
var assets = new List<string>() {"SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
                  "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                  "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"};
```

```
assets = ["SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
          "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
          "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]
```

3. Call the `AddEquity` method with the tickers, and their corresponding resolution.

```
foreach(var ticker in assets){
    qb.AddEquity(ticker, Resolution.Minute);
}
```

```
for i in range(len(assets)):
    qb.AddEquity(assets[i],Resolution.Minute)
```

If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

```
var history = qb.History(qb.Securities.Keys, new DateTime(2021, 1, 1), new DateTime(2021, 12, 31), Resolution.Daily);
```

```
history = qb.History(qb.Securities.Keys, datetime(2021, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
```

Prepare Data

We'll have to process our data to get their correlation and select the least and most related ones.

1. Extract daily return for each `Symbol` from `Slice` data.
2. Select the close column and then call the `unstack` method, then call `pct_change` to compute the daily return.

```
var returns = new Dictionary<string, List<Double>>();
var last = new Dictionary<string, Double>();
foreach(var slice in history){
    foreach(var symbol in slice.Bars.Keys){
        if(!returns.ContainsKey(symbol)){
            returns.Add(symbol, new List<Double>());
            last.Add(symbol, (Double)slice.Bars[symbol].Close);
        }
        var change = (Double) ((Double)slice.Bars[symbol].Close - last[symbol])/last[symbol];
        last[symbol] = (Double)slice.Bars[symbol].Close;
        returns[symbol].Add(change);
    }
}
```

```
returns = history['close'].unstack(level=0).pct_change().iloc[1:]
```

3. Extract daily return for each `Symbol` from `Slice` data.

```
double[,] ret = new double[returns.Values.ElementAt(0).Count - 1, assets.Count];
int k = 0;
foreach(var kvp in returns)
{
    var symbol = kvp.Key;
    for(int i=0; i < returns[symbol].Count - 1; i++)
    {
        ret[i, k] = returns[symbol][i + 1];
    }
    k++;
}
```

4. Write a function to obtain the least and most correlated 5 assets.

```
public Dictionary<string, Double> GetCorrelations(double[,] returns){
    // Get correlation matrix
    var corrMatrix = Measures.Correlation(ret);

    // Find the absolute sum correlation of the assets
    var correlations = new Dictionary<string, Double>();
    for(int i=0; i < corrMatrix.GetLength(0); i++)
    {
        var symbol = assets[i];
        if(!correlations.ContainsKey(symbol)){
            correlations.Add(symbol, 0);
        }
        for (int j=0; j < corrMatrix.GetLength(1); j++)
        {
```

```
            var value_ = corrMatrix[i, j];
            correlations[symbol] += value_ >= 0 ? value_ : -value_;
        }
    }

    return correlations;
}

var corr = GetCorrelations(ret);
var selected = corr.OrderBy(x => x.Value).Take(5);
var benchmark = corr.OrderBy(x => x.Value).TakeLast(5);
```

```
def GetUncorrelatedAssets(returns, num_assets):
    # Get correlation
    correlation = returns.corr()

    # Find assets with lowest and highest absolute sum correlation
    selected = []
    for index, row in correlation.iteritems():
        corr_rank = row.abs().sum()
        selected.append((index, corr_rank))

    # Sort and take the top num_assets
    sort_ = sorted(selected, key = lambda x: x[1])
    uncorrelated = sort_[:num_assets]
    correlated = sort_[-num_assets:]

    return uncorrelated, correlated

selected, benchmark = GetUncorrelatedAssets(returns, 5)
```

Test Hypothesis

To test the hypothesis: Our desired outcome would be a consistent and low fluctuation equity curve should be seen, as compared with benchmark.

1. Construct a equal weighting portfolio for the 5-uncorrelated-asset-portfolio and the 5-correlated-asset-portfolio (benchmark).

```
double[,] portRet = new double[returns.Values.ElementAt(0).Count, 5];
int j = 0;
foreach(var kvp in selected){
    var symbol = kvp.Key;
    for(int i=0; i < returns[symbol].Count; i++)
    {
        portRet[i, j] = returns[symbol][i] / 5;
    }
    j++;
}

double[,] benchRet = new double[returns.Values.ElementAt(0).Count, 5];
j = 0;
foreach(var kvp in benchmark){
    var symbol = kvp.Key;
    for(int i=0; i < returns[symbol].Count; i++)
    {
        benchRet[i, j] = returns[symbol][i] / 5;
    }
    j++;
}
```

```
port_ret = returns[[x[0] for x in selected]] / 5
bench_ret = returns[[x[0] for x in benchmark]] / 5
```

2. Get the Equity Return for both portfolios.
3. Call `cumprod` to get the cumulative return.

```
var totalValue = new List<double>{1.0};
var dailySum = 0.0;
for(int i=0; i < portRet.GetLength(0); i++)
{
    totalValue.Add(totalValue.Last() * (1 + dailySum));
    dailySum = 0.0;
    for (int j=0; j < portRet.GetLength(1); j++)
    {
        if (double.IsFinite(portRet[i, j]))
        {
            dailySum += portRet[i, j];
        }
    }
}

var totalValueBench = new List<double>{1.0};
var dailySumBench = 0.0;
for(int i=0; i < benchRet.GetLength(0); i++)
{
    totalValueBench.Add(totalValueBench.Last() * (1 + dailySumBench));
    dailySumBench = 0.0;
    for (int j=0; j < benchRet.GetLength(1); j++)
    {
        if (double.IsFinite(benchRet[i, j]))
        {
            dailySumBench += benchRet[i, j];
        }
    }
}
```

```
total_ret = (np.sum(port_ret, axis=1) + 1).cumprod()
total_ret_bench = (np.sum(bench_ret, axis=1) + 1).cumprod()
```

4. Calculate the variance of the 2 portfolios.

```
var returnPort = new List<double>();
previous = 0.0;
for(int i=0; i < totalValue.Count; i++)
{
    var current = totalValue[i];
    returnPort.Add((current - previous) / previous);
    previous = current;
}
var varPort = Math.Sqrt(returnPort.Skip(1).Average(v=>Math.Pow(v-returnPort.Skip(1).Average(),2)));

var returnBench = new List<double>();
previous = 0.0;
for(int i=0; i < totalValueBench.Count; i++)
{
    var current = totalValueBench[i];
    returnBench.Add((current - previous) / previous);
    previous = current;
}
var varBench = Math.Sqrt(returnBench.Skip(1).Average(v=>Math.Pow(v-returnBench.Skip(1).Average(),2)));
```

5. Print the result.
6. Plot the result.

```
Console.WriteLine("Portfolio Return: {0}, Variance: {1}", (totalValue.Last() - totalValue.First())/totalValue.First(), varPort);
Console.WriteLine("Benchmark Return: {0}, Variance: {1}", (totalValueBench.Last() - totalValueBench.First())/totalValueBench.First(), varBench);
```

```
plt.figure(figsize=(15, 10))
total_ret.plot(label='Proposed')
```

```
total_ret_bench.plot(label='Benchmark')
plt.title('Equity Curve')
plt.legend()
plt.show()
```

-image

We can clearly see from the results, the proposed uncorrelated-asset-portfolio has a lower variance/fluctuation, thus more consistent than the benchmark. This proven our hypothesis.

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into research is to create a scheduled event which uses our model to pick stocks and goes long.

```
private List<string> _asset = new List<string>{"SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
        "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
        "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"};

public override void Initialize()
{
    // 1. Required: Five years of backtest history
    SetStartDate(2014, 1, 1);

    // 2. Required: Alpha Streams Models:
    SetBrokerageModel(BrokerageName.AlphaStreams);

    // 3. Required: Significant AUM Capacity
    SetCash(1000000);

    // 4. Required: Benchmark to SPY
    SetBenchmark("SPY");

    SetPortfolioConstruction(new EqualWeightingPortfolioConstructionModel());
    SetExecution(new ImmediateExecutionModel());

    // Add Equity ------------------------------------------------
    foreach(var ticker in _asset)
    {
    AddEquity(ticker, Resolution.Minute);
    }

    // Set Scheduled Event Method For Our Model. In this example, we'll rebalance every month.
    Schedule.On(DateRules.MonthStart(),
        TimeRules.BeforeMarketClose("SHY", 5),
        EveryDayBeforeMarketClose);
}
```

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2014, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.SetPortfolioConstruction(EqualWeightingPortfolioConstructionModel())
    self.SetExecution(ImmediateExecutionModel())

    self.assets = ["SHY", "TLT", "IEI", "SHV", "TLH", "EDV", "BIL",
                   "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
                   "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.AddEquity(self.assets[i], Resolution.Minute)

    # Set Scheduled Event Method For Our Model. In this example, we'll rebalance every month.
    self.Schedule.On(self.DateRules.MonthStart(),
        self.TimeRules.BeforeMarketClose("SHY", 5),
        self.EveryDayBeforeMarketClose)
```

Now we export our model into the scheduled event method. We will remove `qb` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm`.

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm`.

```
private void EveryDayBeforeMarketClose()
{
    // Fetch history on our universe
    var history = History(Securities.Keys, 252*2, Resolution.Daily);
    if (history.Count() < 0) return;

    // Extract daily return of close prices for each Symbol from Slice data
    var returns = new Dictionary<string, List<Double>>();
    var last = new Dictionary<string, Double>();
    foreach(var slice in history){
        foreach(var symbol in slice.Bars.Keys){
            if(!returns.ContainsKey(symbol)){
                returns.Add(symbol, new List<Double>());
                last.Add(symbol, (Double)slice.Bars[symbol].Close);
            }
            var change = (Double) ((Double)slice.Bars[symbol].Close - last[symbol])/last[symbol];
            last[symbol] = (Double)slice.Bars[symbol].Close;
            returns[symbol].Add(change);
        }
    }

    // Convert returns into 2-d array
    double[,] ret = new double[returns.Values.ElementAt(0).Count - 1, _asset.Count];
    int k = 0;
    foreach(var kvp in returns)
    {
        var symbol = kvp.Key;
        for(int i=0; i < returns[symbol].Count - 1; i++)
        {
            ret[i, k] = returns[symbol][i + 1];
        }
        k++;
    }

    // Get correlation matrix
    var corrMatrix = Measures.Correlation(ret);

    // Find 5 assets with the least absolute sum correlation
    var correlations = new Dictionary<string, Double>();
    for(int i=0; i < corrMatrix.GetLength(0); i++)
    {
        var symbol = _asset[i];
        if(!correlations.ContainsKey(symbol)){
            correlations.Add(symbol, 0);
```

```
        }
        for (int j=0; j < corrMatrix.GetLength(1); j++)
        {
            var value_ = corrMatrix[i, j];
            correlations[symbol] += value_ >= 0 ? value_ : -value_;
        }
    }
    var selected = correlations.OrderBy(x => x.Value).Take(5).Select(x => x.Key).ToList();

    // Emit insights
    foreach(var symbol in selected)
    {
        var insight = new Insight(symbol, Expiry.EndOfMonth, InsightType.Price, InsightDirection.Up);
        EmitInsights(insight);
    }
}

def EveryDayBeforeMarketClose(self) -> None:
    qb = self
    # Fetch history on our universe
    history = qb.History(qb.Securities.Keys, 252*2, Resolution.Daily)
    if history.empty: return

    # Select the close column and then call the unstack method, then call pct_change to compute the daily return.
    returns = history['close'].unstack(level=0).pct_change().iloc[1:]

    # Get correlation
    correlation = returns.corr()

    # Find 5 assets with lowest absolute sum correlation
    selected = []
    for index, row in correlation.iteritems():
        corr_rank = row.abs().sum()
        selected.append((index, corr_rank))

    sort_ = sorted(selected, key = lambda x: x[1])
    selected = [x[0] for x in sort_[:5]]

    # ==============================

    insights = []

    for symbol in selected:
        insights.append( Insight.Price(symbol, Expiry.EndOfMonth, InsightDirection.Up) )

    self.EmitInsights(insights)
```

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	3047
Average Win	0.02%
Average Loss	0.00%
Compounding Annual Return	-0.364%
Drawdown	3.500%
Expectancy	-0.487
Net Profit	-1.808%
Sharpe Ratio	-0.309
Probabilistic Sharpe Ratio	0.022%
Loss Rate	95%
Win Rate	5%
Profit-Loss Ratio	9.17
Alpha	-0.002
Beta	-0.011
Annual Standard Deviation	0.008
Annual Variance	0
Information Ratio	-0.593
Tracking Error	0.111
Treynor Ratio	0.22
Total Fees	$9690.90
Estimated Strategy Capacity	$470000.00
Lowest Capacity Asset	SST V2245V5VOQQT

```
#region imports
from AlgorithmImports import *
#endregion

class UncorrelatedAssetsDemo(QCAlgorithm):

    def Initialize(self):
        #1. Required: Five years of backtest history
        self.SetStartDate(2014, 1, 1)
        self.SetEndDate(2019, 1, 1)
```

- Charts
- Statistics
- Code
  - Main.cs
  - Research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	4428
Average Win	0.05%
Average Loss	-0.01%
Compounding Annual Return	0.018%
Drawdown	9.300%
Expectancy	0.010
Net Profit	0.091%
Sharpe Ratio	0.023
Probabilistic Sharpe Ratio	0.430%
Loss Rate	79%
Win Rate	21%
Profit-Loss Ratio	3.74
Alpha	0.007
Beta	-0.1
Annual Standard Deviation	0.04
Annual Variance	0.002
Information Ratio	-0.494
Tracking Error	0.127
Treynor Ratio	-0.009
Total Fees	$17169.88
Estimated Strategy Capacity	$290000.00
Lowest Capacity Asset	TLO TT1EBZ21QWKL

```
using Accord.Statistics;
using System.Collections.Generic;
using System.Linq;
using QuantConnect.Algorithm.Framework.Alphas;
using QuantConnect.Algorithm.Framework.Portfolio;
using QuantConnect.Algorithm.Framework.Execution;
using QuantConnect.Brokerages;

namespace QuantConnect.Algorithm.CSharp
{
```

10.5 Kalman Filters and Stat Arb

Introduction

This page explains how to you can use the Research Environment to develop and test a Kalman Filters and Statistical Arbitrage hypothesis, then put the hypothesis in production.

Create Hypothesis

In finance, we can often observe that 2 stocks with similar background and fundamentals (e.g. AAPL vs MSFT, SPY vs QQQ) move in similar manner. They could be correlated, although not necessary, but their price difference/sum (spread) is stationary. We call this cointegration. Thus, we could hypothesize that extreme spread could provide chance for arbitrage, just like a mean reversion of spread. This is known as pairs trading. Likewise, this could also be applied to more than 2 assets, this is known as statistical arbitrage.

However, although the fluctuation of the spread is stationary, the mean of the spread could be changing by time due to different reasons. Thus, it is important to update our expectation on the spread in order to go in and out of the market in time, as the profit margin of this type of short-window trading is tight. Kalman Filter could come in handy in this situation. We can consider it as an updater of the underlying return Markov Chain's expectation, while we're assuming the price series is a Random Process.

In this example, we're making a hypothesis on trading the spread on cointegrated assets is profitable. We'll be using forex pairs EURUSD, GBPUSD, USDCAD, USDHKD and USDJPY for this example, skipping the normalized price difference selection.

Import Libraries

We'll need to import libraries to help with data processing, model building, validation and visualization. Import `arch` , `pykalman` , `scipy` , `statsmodels` , `numpy` , `matplotlib` and `pandas` libraries by the following:

```
from arch.unitroot.cointegration import engle_granger
from pykalman import KalmanFilter
from scipy.optimize import minimize
from statsmodels.tsa.vector_ar.vecm import VECM

import numpy as np
from matplotlib import pyplot as plt
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the desired tickers for research.

   ```
   assets = ["EURUSD", "GBPUSD", "USDCAD", "USDHKD", "USDJPY"]
   ```

3. Call the `AddForex` method with the tickers, and their corresponding resolution. Then store their `Symbol` s.

   ```
   for i in range(len(assets)):
       qb.AddForex(assets[i],Resolution.Minute)
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   history = qb.History(qb.Securities.Keys, datetime(2021, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Cointegration

We'll have to test if the assets are cointegrated. If so, we'll have to obtain the cointegration vector(s).

Cointegration Testing

1. Select the close column and then call the `unstack` method.

   ```
   df = history['close'].unstack(level=0)
   ```

2. Call `np.log` to convert the close price into log-price series to eliminate compounding effect.

   ```
   log_price = np.log(data)
   ```

3. Apply Engle Granger Test to check if the series are cointegrated.

   ```
   coint_result = engle_granger(log_price.iloc[:, 0], log_price.iloc[:, 1:], trend='n', method='bic')
   ```

   It shows a p-value < 0.05 for the unit test, with lag-level 0. This proven the log price series are cointegrated in realtime. The spread of the 5 forex pairs are stationary.

Get Cointegration Vectors

We would use a VECM model to obtain the cointegrated vectors.

1. Initialize a `VECM` model by following the unit test parameters, then fit to our data.

   ```
   vecm_result = VECM(log_price, k_ar_diff=0, coint_rank=len(assets)-1, deterministic='n').fit()
   ```

2. Obtain the `Beta` attribute. This is the cointegration subspaces' unit vectors.

   ```
   beta = vecm_result.beta
   ```

3. Check the spread of different cointegration subspaces.

   ```
   spread = log_price @ beta
   ```

4. Plot the results.

   ```
   fig, axs = plt.subplots(beta.shape[1], figsize=(15, 15))
   fig.suptitle('Spread for various cointegrating vectors')
   for i in range(beta.shape[1]):
       axs[i].plot(spread.iloc[:, i])
       axs[i].set_title(f"The {i+1}th normalized cointegrating subspace")
   plt.show()
   ```

Optimization of Cointegration Subspaces

Although the 4 cointegratoin subspaces are not looking stationarym, we can optimize for a mean-reverting portfolio by putting various weights in different subspaces. We use the Portmanteau statistics as a proxy for the mean reversion. So we formulate:

$$\begin{equation*} \begin{aligned} & \underset{w}{\text{minimize}} & & \mathrm{(\frac {w^{T}M_{1}w} {w^{T}M_{0}w})^{2}} \\ & \text{subject to} & & w^{T}M_{0}w = \nu\\ &&& 1^Tw = 0\\ & \text{where} & & M_i \triangleq Cov(s_t, s_{t+i}) = E[(s_t - E[s_t]) (s_{t+i} - E[s_{t+i}])^T] \\ \end{aligned} \end{equation*}$$

with s is spread, v is predetermined desirable variance level (the larger the higher the profit, but lower the trading frequency)

1. We set the weight on each vector is between -1 and 1. While overall sum is 0.

```
x0 = np.array([-1**i/beta.shape[1] for i in range(beta.shape[1])])
bounds = tuple((-1, 1) for i in range(beta.shape[1]))
constraints = [{'type':'eq', 'fun':lambda x: np.sum(x)}]
```

2. Optimize the Portmanteau statistics.

```
opt = minimize(lambda w: ((w.T @ np.cov(spread.T, spread.shift(1).fillna(0).T)[spread.shape[1]:, :spread.shape[1]] @ w)/(w.T @ np.cov(spread.T) @ w))**2,
               x0=x0,
               bounds=bounds,
               constraints=constraints,
               method="SLSQP")
```

3. Normalize the result.

```
opt.x = opt.x/np.sum(abs(opt.x))
for i in range(len(opt.x)):
    print(f"The weight put on {i+1}th normalized cointegrating subspace: {opt.x[i]}")
```

4. Plot the weighted spread.

```
new_spread = spread @ opt.x
new_spread.plot(title="Weighted spread", figsize=(15, 10))
plt.ylabel("Spread")
plt.show()
```

Kalman Filter

The weighted spread looks more stationary. However, the fluctuation half-life is very long accrossing zero. We aim to trade as much as we can to maximize the profit of this strategy. Kalman Filter then comes into the play. It could modify the expectation of the next step based on smoothening the prediction and actual probability distribution of return.

Image Source: Understanding Kalman Filters, Part 3: An Optimal State Estimator. Melda Ulusoy (2017). MathWorks. Retreived from: https://www.mathworks.com/videos/understanding-kalman-filters-part-3-optimal-state-estimator--1490710645421.html

1. Initialize a `KalmanFilter` .

   In this example, we use the first 20 data points to optimize its initial state. We assume the market has no regime change so that the transitional matrix and observation matrix is [1].

```
kalmanFilter = KalmanFilter(transition_matrices = [1],
                  observation_matrices = [1],
                  initial_state_mean = new_spread.iloc[:20].mean(),
                  observation_covariance = new_spread.iloc[:20].var(),
                  em_vars=['transition_covariance', 'initial_state_covariance'])
kalmanFilter = kalmanFilter.em(new_spread.iloc[:20], n_iter=5)
(filtered_state_means, filtered_state_covariances) = kalmanFilter.filter(new_spread.iloc[:20])
```

2. Obtain the current Mean and Covariance Matrix expectations.

```
currentMean = filtered_state_means[-1, :]
currentCov = filtered_state_covariances[-1, :]
```

3. Initialize a mean series for spread normalization using the `KalmanFilter` 's results.

```
mean_series = np.array([None]*(new_spread.shape[0]-100))
```

4. Roll over the Kalman Filter to obtain the mean series.

```
for i in range(100, new_spread.shape[0]):
    (currentMean, currentCov) = kalmanFilter.filter_update(filtered_state_mean = currentMean,
                                                           filtered_state_covariance = currentCov,
                                                           observation = new_spread.iloc[i])
    mean_series[i-100] = float(currentMean)
```

5. Obtain the normalized spread series.

```
normalized_spread = (new_spread.iloc[100:] - mean_series)
```

6. Plot the normalized spread series.

```
plt.figure(figsize=(15, 10))
plt.plot(normalized_spread, label="Processed spread")
plt.title("Normalized spread series")
plt.ylabel("Spread - Expectation")
plt.legend()
plt.show()
```

Determine Trading Threshold

Now we need to determine the threshold of entry. We want to maximize profit from each trade (variance of spread) x frequency of entry. To do so, we formulate:

$$\begin{aligned} & \underset{f}{\text{minimize}} & & \begin{Vmatrix} \bar{f} - f \end{Vmatrix}_{2}^{2} + \lambda\ \begin{Vmatrix} Df \end{Vmatrix}_{2}^{2} \\ & \text{where} & & \bar{f_j} = \frac{\sum_{t=1}^T 1_{\{spread_t\ >\ set\ level_j\}}}{T} \\ &&& D = \begin{bmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \end{bmatrix} \in \mathbb{R}^{(j - 1) \times j} \end{aligned}$$

so $f^* = (I+\lambda D^TD)^{-1}\bar{f}$

1. Initialize 50 set levels for testing.

```
s0 = np.linspace(0, max(normalized_spread), 50)
```

2. Calculate the profit levels using the 50 set levels.

```
f_bar = np.array([None]*50)
for i in range(50):
    f_bar[i] = len(normalized_spread.values[normalized_spread.values > s0[i]]) / normalized_spread.shape[0]
```

3. Set trading frequency matrix.

```
D = np.zeros((49, 50))
for i in range(D.shape[0]):
    D[i, i] = 1
    D[i, i+1] = -1
```

4. Set level of lambda.

```
l = 1.0
```

5. Obtain the normalized profit level.

```
f_star = np.linalg.inv(np.eye(50) + l * D.T@D) @ f_bar.reshape(-1, 1)
s_star = [f_star[i]*s0[i] for i in range(50)]
```

6. Get the maximum profit level as threshold.

```
threshold = s0[s_star.index(max(s_star))]
print(f"The optimal threshold is {threshold}")
```

7. Plot the result.

```
plt.figure(figsize=(15, 10))
plt.plot(s0, s_star)
plt.title("Profit of mean-revertion trading")
plt.xlabel("Threshold")
```

```
plt.ylabel("Profit")
plt.show()
```

Test Hypothesis

To test the hypothesis. We wish to obtain a profiting strategy.

1. Set the trading weight. We would like the portfolio absolute total weight is 1 when trading.

```
trading_weight = beta @ opt.x
trading_weight /= np.sum(abs(trading_weight))
```

2. Set up the trading data.

```
testing_ret = data.pct_change().iloc[1:].shift(-1)   # Shift 1 step backward as forward return result
equity = pd.DataFrame(np.ones((testing_ret.shape[0], 1)), index=testing_ret.index, columns=["Daily value"])
```

3. Set the buy and sell preiod when the spread exceeds the threshold.

```
buy_period = normalized_spread[normalized_spread < -threshold].index
sell_period = normalized_spread[normalized_spread > threshold].index
```

4. Trade the portfolio.

```
equity.loc[buy_period, "Daily value"] = testing_ret.loc[buy_period] @ trading_weight + 1
equity.loc[sell_period, "Daily value"] = testing_ret.loc[sell_period] @ -trading_weight + 1
```

5. Get the total portfolio value.

```
value = equity.cumprod()
```

6. Plot the result.

```
value.plot(title="Equity Curve", figsize=(15, 10))
plt.ylabel("Portfolio Value")
plt.show()
```

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into backtest is to create a scheduled event which uses our model to predict the expected return.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2014, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.assets = ["EURUSD", "GBPUSD", "USDCAD", "USDHKD", "USDJPY"]

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.AddForex(self.assets[i], Resolution.Minute)

    # Instantiate our model
    self.Recalibrate()

    # Set a variable to indicate the trading bias of the portfolio
    self.state = 0

    # Set Scheduled Event Method For Recalibrate Our Model Every Week.
    self.Schedule.On(self.DateRules.WeekStart(),
        self.TimeRules.At(0, 0),
        self.Recalibrate)

    # Set Scheduled Event Method For Kalman Filter updating.
    self.Schedule.On(self.DateRules.EveryDay(),
        self.TimeRules.BeforeMarketClose("EURUSD"),
        self.EveryDayBeforeMarketClose)
```

We'll also need to create a function to train and update our model from time to time. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
def Recalibrate(self) -> None:
    qb = self
    history = qb.History(self.assets, 252*2, Resolution.Daily)
    if history.empty: return

    # Select the close column and then call the unstack method
    data = history['close'].unstack(level=0)

    # Convert into log-price series to eliminate compounding effect
    log_price = np.log(data)

    ### Get Cointegration Vectors
    # Initialize a VECM model following the unit test parameters, then fit to our data.
    vecm_result = VECM(log_price, k_ar_diff=0, coint_rank=len(self.assets)-1, deterministic='n').fit()

    # Obtain the Beta attribute. This is the cointegration subspaces' unit vectors.
    beta = vecm_result.beta

    # Check the spread of different cointegration subspaces.
    spread = log_price @ beta

    ### Optimization of Cointegration Subspaces
    # We set the weight on each vector is between -1 and 1. While overall sum is 0.
    x0 = np.array([-1**i/beta.shape[1] for i in range(beta.shape[1])])
    bounds = tuple((-1, 1) for i in range(beta.shape[1]))
    constraints = [{'type':'eq', 'fun':lambda x: np.sum(x)}]

    # Optimize the Portmanteau statistics
    opt = minimize(lambda w: ((w.T @ np.cov(spread.T, spread.shift(1).fillna(0).T)[spread.shape[1]:, :spread.shape[1]] @ w)/(w.T @ np.cov(spread.T) @ w))**2,
                   x0=x0,
                   bounds=bounds,
                   constraints=constraints,
                   method="SLSQP")

    # Normalize the result
    opt.x = opt.x/np.sum(abs(opt.x))
    new_spread = spread @ opt.x

    ### Kalman Filter
    # Initialize a Kalman Filter. Using the first 20 data points to optimize its initial state. We assume the market has no regime change so that the transitional matrix and obse
    self.kalmanFilter = KalmanFilter(transition_matrices = [1],
                        observation_matrices = [1],
                        initial_state_mean = new_spread.iloc[:20].mean(),
                        observation_covariance = new_spread.iloc[:20].var(),
                        em_vars=['transition_covariance', 'initial_state_covariance'])
```

```
    self.kalmanFilter = self.kalmanFilter.em(new_spread.iloc[:20], n_iter=5)
    (filtered_state_means, filtered_state_covariances) = self.kalmanFilter.filter(new_spread.iloc[:20])

    # Obtain the current Mean and Covariance Matrix expectations.
    self.currentMean = filtered_state_means[-1, :]
    self.currentCov = filtered_state_covariances[-1, :]

    # Initialize a mean series for spread normalization using the Kalman Filter's results.
    mean_series = np.array([None]*(new_spread.shape[0]-20))

    # Roll over the Kalman Filter to obtain the mean series.
    for i in range(20, new_spread.shape[0]):
        (self.currentMean, self.currentCov) = self.kalmanFilter.filter_update(filtered_state_mean = self.currentMean,
                                                            filtered_state_covariance = self.currentCov,
                                                            observation = new_spread.iloc[i])
        mean_series[i-20] = float(self.currentMean)

    # Obtain the normalized spread series.
    normalized_spread = (new_spread.iloc[20:] - mean_series)

    ### Determine Trading Threshold
    # Initialize 50 set levels for testing.
    s0 = np.linspace(0, max(normalized_spread), 50)

    # Calculate the profit levels using the 50 set levels.
    f_bar = np.array([None]*50)
    for i in range(50):
        f_bar[i] = len(normalized_spread.values[normalized_spread.values > s0[i]]) \
            / normalized_spread.shape[0]

    # Set trading frequency matrix.
    D = np.zeros((49, 50))
    for i in range(D.shape[0]):
        D[i, i] = 1
        D[i, i+1] = -1

    # Set level of lambda.
    l = 1.0

    # Obtain the normalized profit level.
    f_star = np.linalg.inv(np.eye(50) + l * D.T@D) @ f_bar.reshape(-1, 1)
    s_star = [f_star[i]*s0[i] for i in range(50)]
    self.threshold = s0[s_star.index(max(s_star))]

    # Set the trading weight. We would like the portfolio absolute total weight is 1 when trading.
    trading_weight = beta @ opt.x
    self.trading_weight = trading_weight / np.sum(abs(trading_weight))
```

Now we export our model into the scheduled event method for trading. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm`.

```
def EveryDayBeforeMarketClose(self) -> None:
    qb = self

    # Get the real-time log close price for all assets and store in a Series
    series = pd.Series()
    for symbol in qb.Securities.Keys:
        series[symbol] = np.log(qb.Securities[symbol].Close)

    # Get the spread
    spread = series @ self.trading_weight

    # Update the Kalman Filter with the Series
    (self.currentMean, self.currentCov) = self.kalmanFilter.filter_update(filtered_state_mean = self.currentMean,
                                                                    filtered_state_covariance = self.currentCov,
                                                                    observation = spread)

    # Obtain the normalized spread.
    normalized_spread = spread - self.currentMean

    # ==============================

    # Mean-reversion
    if normalized_spread < -self.threshold:
        orders = []
        for i in range(len(self.assets)):
            orders.append(PortfolioTarget(self.assets[i], self.trading_weight[i]))
            self.SetHoldings(orders)

        self.state = 1

    elif normalized_spread > self.threshold:
        orders = []
        for i in range(len(self.assets)):
            orders.append(PortfolioTarget(self.assets[i], -1 * self.trading_weight[i]))
            self.SetHoldings(orders)

        self.state = -1

    # Out of position if spread recovered
    elif self.state == 1 and normalized_spread > -self.threshold or self.state == -1 and normalized_spread < self.threshold:
        self.Liquidate()

        self.state = 0
```

Reference

1. A Signal Processing Perspective on Financial Engineering. Y. Feng, D. P. Palomer (2016). *Foundations and Trends in Signal Processing. 9(1-2). p173-200.*

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

QUANTCONNECT

Overall Statistics	
Total Trades	1315
Average Win	0.06%
Average Loss	-0.07%
Compounding Annual Return	-3.477%
Drawdown	17.000%
Expectancy	-0.349
Net Profit	-16.232%
Sharpe Ratio	-1.024
Probabilistic Sharpe Ratio	0.000%
Loss Rate	64%
Win Rate	36%
Profit-Loss Ratio	0.79
Alpha	-0.025
Beta	0.022
Annual Standard Deviation	0.023
Annual Variance	0.001
Information Ratio	-0.796
Tracking Error	0.11
Treynor Ratio	-1.088
Total Fees	$109.25
Estimated Strategy Capacity	$43000000.00
Lowest Capacity Asset	USDHKD 8G

```
#region imports
from AlgorithmImports import *
from pykalman import KalmanFilter
from scipy.optimize import minimize
from statsmodels.tsa.vector_ar.vecm import VECM
#endregion

class KalmanFilterStatisticalArbitrageDemo(QCAlgorithm):

    def Initialize(self):
```

10.6 PCA and Pairs Trading

Introduction

This page explains how to you can use the Research Environment to develop and test a Principle Component Analysis hypothesis, then put the hypothesis in production.

Create Hypothesis

Principal Component Analysis (PCA) a way of mapping the existing dataset into a new "space", where the dimensions of the new data are linearly-independent, orthogonal vectors. PCA eliminates the problem of multicollinearity. In another way of thought, can we actually make use of the collinearity it implied, to find the collinear assets to perform pairs trading?

Import Libraries

We'll need to import libraries to help with data processing, validation and visualization. Import `sklearn` , `arch` , `statsmodels` , `numpy` and `matplotlib` libraries by the following:

```
from sklearn.decomposition import PCA
from arch.unitroot.cointegration import engle_granger
from statsmodels.tsa.stattools import adfuller
import numpy as np
from matplotlib import pyplot as plt
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the desired tickers for research.

   ```
   symbols = {}
   assets = ["SHY", "TLT", "SHV", "TLH", "EDV", "BIL",
             "SPTL", "TBT", "TMF", "TMV", "TBF", "VGSH", "VGIT",
             "VGLT", "SCHO", "SCHR", "SPTS", "GOVT"]
   ```

3. Call the `AddEquity` method with the tickers, and their corresponding resolution. Then store their `Symbol` s.

   ```
   for i in range(len(assets)):
       symbols[assets[i]] = qb.AddEquity(assets[i],Resolution.Minute).Symbol
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   history = qb.History(qb.Securities.Keys, datetime(2021, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data to get the principle component unit vector that explains the most variance, then find the highest- and lowest-absolute-weighing assets as the pair, since the lowest one's variance is mostly explained by the highest.

1. Select the close column and then call the `unstack` method.

   ```
   close_price = history['close'].unstack(level=0)
   ```

2. Call `pct_change` to compute the daily return.

   ```
   returns = close_price.pct_change().iloc[1:]
   ```

3. Initialize a `PCA` model, then get the principle components by the maximum likelihood.

   ```
   pca = PCA()
   pca.fit(returns)
   ```

4. Get the number of principle component in a list, and their corresponding explained variance ratio.

   ```
   components = [str(x + 1) for x in range(pca.n_components_)]
   explained_variance_pct = pca.explained_variance_ratio_ * 100
   ```

5. Plot the principle components' explained variance ratio.

   ```
   plt.figure(figsize=(15, 10))
   plt.bar(components, explained_variance_pct)
   plt.title("Ratio of Explained Variance")
   plt.xlabel("Principle Component #")
   plt.ylabel("%")
   plt.show()
   ```

   We can see over 95% of the variance is explained by the first principle. We could conclude that collinearity exists and most assets' return are correlated. Now, we can extract the 2 most correlated pairs.

6. Get the weighting of each asset in the first principle component.

   ```
   first_component = pca.components_[0, :]
   ```

7. Select the highest- and lowest-absolute-weighing asset.

   ```
   highest = assets[abs(first_component).argmax()]
   lowest = assets[abs(first_component).argmin()]
   print(f'The highest-absolute-weighing asset: {highest}\nThe lowest-absolute-weighing asset: {lowest}')
   ```

8. Plot their weighings.

   ```
   plt.figure(figsize=(15, 10))
   plt.bar(assets, first_component)
   plt.title("Weightings of each asset in the first component")
   plt.xlabel("Assets")
   plt.ylabel("Weighting")
   plt.xticks(rotation=30)
   plt.show()
   ```

Test Hypothesis

We now selected 2 assets as candidate for pair-trading. Hence, we're going to test if they are cointegrated and their spread is stationary to do so.

1. Call `np.log` to get the log price of the pair.

   ```
   log_price = np.log(close_price[[highest, lowest]])
   ```

2. Test cointegration by Engle Granger Test.

   ```
   coint_result = engle_granger(log_price.iloc[:, 0], log_price.iloc[:, 1], trend="c", lags=0)
   display(coint_result)
   ```

3. Get their cointegrating vector.

   ```
   coint_vector = coint_result.cointegrating_vector[:2]
   ```

4. Calculate the spread.

```
spread = log_price @ coint_vector
```

5. Use Augmented Dickey Fuller test to test its stationarity.

```
pvalue = adfuller(spread, maxlag=0)[1]
print(f"The ADF test p-value is {pvalue}, so it is {'' if pvalue < 0.05 else 'not '}stationary.")
```

6. Plot the spread.

```
spread.plot(figsize=(15, 10), title=f"Spread of {highest} and {lowest}")
plt.ylabel("Spread")
plt.show()
```

Result shown that the pair is cointegrated and their spread is stationary, so they are potential pair for pair-trading.

Set Up Algorithm

Pairs trading is exactly a 2-asset version of statistical arbitrage. Thus, we can just modify the algorithm from [the Kalman Filter and Statistical Arbitrage tutorial](), except we're using only a single cointegrating unit vector so no optimization of cointegration subspace is needed.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2014, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.assets = ["SCHO", "SHY"]

    # Add Equity ------------------------------------------------
    for i in range(len(self.assets)):
        self.AddEquity(self.assets[i], Resolution.Minute)

    # Instantiate our model
    self.Recalibrate()

    # Set a variable to indicate the trading bias of the portfolio
    self.state = 0

    # Set Scheduled Event Method For Kalman Filter updating.
    self.Schedule.On(self.DateRules.WeekStart(),
        self.TimeRules.At(0, 0),
        self.Recalibrate)

    # Set Scheduled Event Method For Kalman Filter updating.
    self.Schedule.On(self.DateRules.EveryDay(),
        self.TimeRules.BeforeMarketClose("SHY"),
        self.EveryDayBeforeMarketClose)


def Recalibrate(self) -> None:
    qb = self
    history = qb.History(self.assets, 252*2, Resolution.Daily)
    if history.empty: return

    # Select the close column and then call the unstack method
    data = history['close'].unstack(level=0)

    # Convert into log-price series to eliminate compounding effect
    log_price = np.log(data)

    ### Get Cointegration Vectors
    # Get the cointegration vector
    coint_result = engle_granger(log_price.iloc[:, 0], log_price.iloc[:, 1], trend="c", lags=0)
    coint_vector = coint_result.cointegrating_vector[:2]

    # Get the spread
    spread = log_price @ coint_vector

    ### Kalman Filter
    # Initialize a Kalman Filter. Using the first 20 data points to optimize its initial state. We assume the market has no regime change so that the transitional matrix and obse
    self.kalmanFilter = KalmanFilter(transition_matrices = [1],
                        observation_matrices = [1],
                        initial_state_mean = spread.iloc[:20].mean(),
                        observation_covariance = spread.iloc[:20].var(),
                        em_vars=['transition_covariance', 'initial_state_covariance'])
    self.kalmanFilter = self.kalmanFilter.em(spread.iloc[:20], n_iter=5)
    (filtered_state_means, filtered_state_covariances) = self.kalmanFilter.filter(spread.iloc[:20])

    # Obtain the current Mean and Covariance Matrix expectations.
    self.currentMean = filtered_state_means[-1, :]
    self.currentCov = filtered_state_covariances[-1, :]

    # Initialize a mean series for spread normalization using the Kalman Filter's results.
    mean_series = np.array([None]*(spread.shape[0]-20))

    # Roll over the Kalman Filter to obtain the mean series.
    for i in range(20, spread.shape[0]):
        (self.currentMean, self.currentCov) = self.kalmanFilter.filter_update(filtered_state_mean = self.currentMean,
                                                            filtered_state_covariance = self.currentCov,
                                                            observation = spread.iloc[i])
        mean_series[i-20] = float(self.currentMean)

    # Obtain the normalized spread series.
    normalized_spread = (spread.iloc[20:] - mean_series)

    ### Determine Trading Threshold
    # Initialize 50 set levels for testing.
    s0 = np.linspace(0, max(normalized_spread), 50)

    # Calculate the profit levels using the 50 set levels.
    f_bar = np.array([None]*50)
    for i in range(50):
        f_bar[i] = len(normalized_spread.values[normalized_spread.values > s0[i]]) \
            / normalized_spread.shape[0]

    # Set trading frequency matrix.
    D = np.zeros((49, 50))
    for i in range(D.shape[0]):
        D[i, i] = 1
        D[i, i+1] = -1

    # Set level of lambda.
    l = 1.0

    # Obtain the normalized profit level.
    f_star = np.linalg.inv(np.eye(50) + l * D.T@D) @ f_bar.reshape(-1, 1)
    s_star = [f_star[i]*s0[i] for i in range(50)]
```

```
    self.threshold = s0[s_star.index(max(s_star))]

    # Set the trading weight. We would like the portfolio absolute total weight is 1 when trading.
    self.trading_weight = coint_vector / np.sum(abs(coint_vector))


def EveryDayBeforeMarketClose(self) -> None:
    qb = self

    # Get the real-time log close price for all assets and store in a Series
    series = pd.Series()
    for symbol in qb.Securities.Keys:
        series[symbol] = np.log(qb.Securities[symbol].Close)

    # Get the spread
    spread = np.sum(series * self.trading_weight)

    # Update the Kalman Filter with the Series
    (self.currentMean, self.currentCov) = self.kalmanFilter.filter_update(filtered_state_mean = self.currentMean,
                                                                          filtered_state_covariance = self.currentCov,
                                                                          observation = spread)

    # Obtain the normalized spread.
    normalized_spread = spread - self.currentMean

    # ==============================

    # Mean-reversion
    if normalized_spread < -self.threshold:
        orders = []
        for i in range(len(self.assets)):
            orders.append(PortfolioTarget(self.assets[i], self.trading_weight[i]))
            self.SetHoldings(orders)

        self.state = 1

    elif normalized_spread > self.threshold:
        orders = []
        for i in range(len(self.assets)):
            orders.append(PortfolioTarget(self.assets[i], -1 * self.trading_weight[i]))
            self.SetHoldings(orders)

        self.state = -1

    # Out of position if spread recovered
    elif self.state == 1 and normalized_spread > -self.threshold or self.state == -1 and normalized_spread < self.threshold:
        self.Liquidate()

        self.state = 0
```

Clone Example Project

- Charts
- Statistics
- Code
    - main.py
    - research.ipynb

Clone Algorithm

Overall Statistics

Total Trades	4118
Average Win	0.00%
Average Loss	0.00%
Compounding Annual Return	-0.050%
Drawdown	0.500%
Expectancy	-0.078
Net Profit	-0.453%
Sharpe Ratio	-0.158
Probabilistic Sharpe Ratio	0.001%
Loss Rate	69%
Win Rate	31%
Profit-Loss Ratio	1.96
Alpha	-0
Beta	0.001
Annual Standard Deviation	0.002
Annual Variance	0
Information Ratio	-0.589
Tracking Error	0.149
Treynor Ratio	-0.325
Total Fees	$4693.25
Estimated Strategy Capacity	$1200000.00
Lowest Capacity Asset	SCHO UOVIOSUIT3DX

```
#region imports
from AlgorithmImports import *
from arch.unitroot.cointegration import engle_granger
from pykalman import KalmanFilter
#endregion

class PCADemo(QCAlgorithm):

    def Initialize(self):
```

10.7 Hidden Markov Models

Introduction

This page explains how to you can use the Research Environment to develop and test a Hidden Markov Model hypothesis, then put the hypothesis in production.

Create Hypothesis

A Markov process is a stochastic process where the possibility of switching to another state depends only on the current state of the model by the current state's probability distribution (it is usually represented by a state transition matrix). It is history-independent, or memoryless. While often a Markov process's state is observable, the states of a Hidden Markov Model (HMM) is not observable. This means the input(s) and output(s) are observable, but their intermediate, the state, is non-observable/hidden.

A 3-state HMM example, where S are the hidden states, O are the observable states and a are the probabilities of state transition.
Image source: Modeling Strategic Use of Human Computer Interfaces with Novel Hidden Markov Models. L. J. Mariano, et. al. (2015). Frontiers in Psychology 6:919. DOI:10.3389/fpsyg.2015.00919

In finance, HMM is particularly useful in determining the market regime, usually classified into "Bull" and "Bear" markets. Another popular classification is "Volatile" vs "Involatile" market, such that we can avoid entering the market when it is too risky. We hypothesis a HMM could be able to do the later, so we can produce a SPY-out-performing portfolio (positive alpha).

Import Libraries

We'll need to import libraries to help with data processing, validation and visualization. Import `statsmodels` , `scipy` , `numpy` , `matplotlib` and `pandas` libraries by the following:

```
from statsmodels.tsa.regime_switching.markov_regression import MarkovRegression
from scipy.stats import multivariate_normal
import numpy as np

from matplotlib import pyplot as plt
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the desired index for research.

   ```
   asset = "SPX"
   ```

3. Call the `AddIndex` method with the tickers, and their corresponding resolution.

   ```
   qb.AddIndex(asset, Resolution.Minute)
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   history = qb.History(qb.Securities.Keys, datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data to get the volatility of the market for classification.

1. Select the close column and then call the `unstack` method.

   ```
   close_price = history['close'].unstack(level=0)
   ```

2. Call `pct_change` to compute the daily return.

   ```
   returns = close_price.pct_change().iloc[1:]
   ```

3. Initialize the HMM, then fit by the daily return data. Note that we're using varinace as switching regime, so `switching_variance` argument is set as `True` .

   ```
   model = MarkovRegression(returns, k_regimes=2, switching_variance=True).fit()
   display(model.summary())
   ```

   All p-values of the regime self-transition coefficients and the regime transition probability matrix's coefficient is smaller than 0.05, indicating the model should be able to classify the data into 2 different volatility regimes.

Test Hypothesis

We now verify if the model can detect high and low volatility period effectively.

1. Get the regime as a column, 1 as Low Variance Regime, 2 as High Variance Regime.

   ```
   regime = pd.Series(model.smoothed_marginal_probabilities.values.argmax(axis=1)+1,
                         index=returns.index, name='regime')
   df_1 = close.loc[returns.index][regime == 1]
   df_2 = close.loc[returns.index][regime == 2]
   ```

2. Get the mean and covariance matrix of the 2 regimes, assume 0 covariance between the two.

   ```
   mean = np.array([returns.loc[df_1.index].mean(), returns.loc[df_2.index].mean()])
   cov = np.array([[returns.loc[df_1.index].var(), 0], [0, returns.loc[df_2.index].var()]])
   ```

3. Fit a 2-dimensional multivariate normal distribution by the 2 means and covriance matrix.

   ```
   dist = multivariate_normal(mean=mean.flatten(), cov=cov)
   mean_1, mean_2 = mean[0], mean[1]
   sigma_1, sigma_2 = cov[0,0], cov[1,1]
   ```

4. Get the normal distribution of each of the distribution.

   ```
   x = np.linspace(-0.05, 0.05, num=100)
   y = np.linspace(-0.05, 0.05, num=100)
   X, Y = np.meshgrid(x,y)
   pdf = np.zeros(X.shape)
   for i in range(X.shape[0]):
       for j in range(X.shape[1]):
           pdf[i,j] = dist.pdf([X[i,j], Y[i,j]])
   ```

5. Plot the probability of data in different regimes.

   ```
   fig, axes = plt.subplots(2, figsize=(15, 10))
   ax = axes[0]
   ax.plot(model.smoothed_marginal_probabilities[0])
   ax.set(title='Smoothed probability of Low Variance Regime')
   ax = axes[1]
   ax.plot(model.smoothed_marginal_probabilities[1])
   ax.set(title='Smoothed probability of High Variance Regime')
   fig.tight_layout()
   plt.show()
   ```

6. Plot the series into regime-wise.

```
df_1.index = pd.to_datetime(df_1.index)
df_1 = df_1.sort_index()
df_2.index = pd.to_datetime(df_2.index)
df_2 = df_2.sort_index()
plt.figure(figsize=(15, 10))
plt.scatter(df_1.index, df_1, color='blue', label="Low Variance Regime")
plt.scatter(df_2.index, df_2, color='red', label="High Variance Regime")
plt.title("Price series")
plt.ylabel("Price ($)")
plt.xlabel("Date")
plt.legend()
plt.show()
```

7. Plot the distribution surface.

```
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(122, projection = '3d')
ax.plot_surface(X, Y, pdf, cmap = 'viridis')
ax.axes.zaxis.set_ticks([])
plt.xlabel("Low Volatility Regime")
plt.ylabel("High Volatility Regime")
plt.title('Bivariate normal distribution of the Regimes')
plt.tight_layout()
plt.show()
```

8. Plot the contour.

```
plt.figure(figsize=(12, 8))
plt.contourf(X, Y, pdf, cmap = 'viridis')
plt.xlabel("Low Volatility Regime")
plt.ylabel("High Volatility Regime")
plt.title('Bivariate normal distribution of the Regimes')
plt.tight_layout()
plt.show()
```

We can clearly seen from the results, the Low Volatility Regime has much lower variance than the High Volatility Regime, proven the model works.

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into backtest is to create a scheduled event which uses our model to predict the expected return. Since we could calculate the expected return, we'd use Mean-Variance Optimization for portfolio construction.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2008, 1, 1)
    self.SetEndDate(2021, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.assets = ["SPY", "TLT"]    # "TLT" as fix income in out-of-market period (high volatility)

    # Add Equity ------------------------------------------------
    for ticker in self.assets:
        self.AddEquity(ticker, Resolution.Minute)

    # Set Scheduled Event Method For Kalman Filter updating.
    self.Schedule.On(self.DateRules.EveryDay(),
        self.TimeRules.BeforeMarketClose("SPY", 5),
        self.EveryDayBeforeMarketClose)
```

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm`.

```
def EveryDayBeforeMarketClose(self) -> None:
    qb = self

    # Get history
    history = qb.History(["SPY"], datetime(2010, 1, 1), datetime.now(), Resolution.Daily)

    # Get the close price daily return.
    close = history['close'].unstack(level=0)

    # Call pct_change to obtain the daily return
    returns = close.pct_change().iloc[1:]

    # Initialize the HMM, then fit by the standard deviation data.
    model = MarkovRegression(returns, k_regimes=2, switching_variance=True).fit()

    # Obtain the market regime
    regime = model.smoothed_marginal_probabilities.values.argmax(axis=1)[-1]

    # ==============================

    if regime == 0:
        self.SetHoldings([PortfolioTarget("TLT", 0.), PortfolioTarget("SPY", 1.)])
    else:
        self.SetHoldings([PortfolioTarget("TLT", 1.), PortfolioTarget("SPY", 0.)])
```

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

-

Overall Statistics	
Total Trades	543
Average Win	2.03%
Average Loss	-0.86%
Compounding Annual Return	10.462%
Drawdown	25.500%
Expectancy	0.495
Net Profit	264.919%
Sharpe Ratio	0.637
Probabilistic Sharpe Ratio	4.698%
Loss Rate	55%
Win Rate	45%
Profit-Loss Ratio	2.34
Alpha	0.076
Beta	0.037
Annual Standard Deviation	0.125
Annual Variance	0.016
Information Ratio	-0.011
Tracking Error	0.207
Treynor Ratio	2.122
Total Fees	$31659.21
Estimated Strategy Capacity	$47000000.00
Lowest Capacity Asset	SPY R735QTJ8XC9X

- QUANTCONNECT

```
9#region imports
from AlgorithmImports import *
from statsmodels.tsa.regime_switching.markov_regression import MarkovRegression
#endregion

class HMMDemo(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2008, 1, 1)
        self.SetEndDate(2021, 1, 1)
```

10.8 Long Short-Term Memory

Introduction

This page explains how to you can use the Research Environment to develop and test a Long Short Term Memory hypothesis, then put the hypothesis in production.

Recurrent neural networks (RNN) are a powerful tool in deep learning. These models quite accurately mimic how humans process sequencial information and learn. Unlike traditional feedforward neural networks, RNNs have memory. That is, information fed into them persists and the network is able to draw on this to make inferences.

Long Short-term Memory (LSTM) is a type of RNN. Instead of one layer, LSTM cells generally have four, three of which are part of "gates" -- ways to optionally let information through. The three gates are commonly referred to as the forget, input, and output gates. The forget gate layer is where the model decides what information to keep from prior states. At the input gate layer, the model decides which values to update. Finally, the output gate layer is where the final output of the cell state is decided. Essentially, LSTM separately decides what to remember and the rate at which it should update.

An exmaple of a LSTM cell: x is the input data, c is the long-term memory, h is the current state and serve as short-term memory, σ and $tanh$ is the non-linear activation function of the gates.
Image source: https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:LSTM_Cell.svg

Create Hypothesis

LSTM models have produced some great results when applied to time-series prediction. One of the central challenges with conventional time-series models is that, despite trying to account for trends or other non-stationary elements, it is almost impossible to truly predict an outlier like a recession, flash crash, liquidity crisis, etc. By having a long memory, LSTM models are better able to capture these difficult trends in the data without suffering from the level of overfitting a conventional model would need in order to capture the same data.

For a very basic application, we're hypothesizing LSTM can offer an accurate prediction in future price.

Import Libraries

We'll need to import libraries to help with data processing, validation and visualization. Import `keras` , `sklearn` , `numpy` and `matplotlib` libraries by the following:

```
from keras.layers import LSTM, Dense, Dropout
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import MinMaxScaler

import numpy as np
from matplotlib import pyplot as plt
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the desired index for research.

   ```
   asset = "SPY"
   ```

3. Call the `AddEquity` method with the tickers, and their corresponding resolution.

   ```
   qb.AddEquity(asset, Resolution.Minute)
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with `qb.Securities.Keys` for all tickers, time argument(s), and resolution to request historical data for the symbol.

   ```
   history = qb.History(qb.Securities.Keys, datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data as well as build the LSTM model before testing the hypothesis. We would scale our data to for better covergence.

1. Select the close column and then call the `unstack` method.

   ```
   close_price = history['close'].unstack(level=0)
   ```

2. Initialize `MinMaxScaler` to scale the data onto [0,1].

   ```
   scaler = MinMaxScaler(feature_range = (0, 1))
   ```

3. Transform our data.

   ```
   df = pd.DataFrame(scaler.fit_transform(close), index=close.index)
   ```

4. Select input data

   ```
   scaler = MinMaxScaler(feature_range = (0, 1))
   ```

5. Shift the data for 1-step backward as training output result.

   ```
   output = df.shift(-1).iloc[:-1]
   ```

6. Split the data into training and testing sets.

   In this example, we use the first 80% data for trianing, and the last 20% for testing.

   ```
   splitter = int(input_.shape[0] * 0.8)
   X_train = input_.iloc[:splitter]
   X_test = input_.iloc[splitter:]
   y_train = output.iloc[:splitter]
   y_test = output.iloc[splitter:]
   ```

7. Build feauture and label sets (using number of steps 60, and feature rank 1).

   ```
   features_set = []
   labels = []
   for i in range(60, X_train.shape[0]):
       features_set.append(X_train.iloc[i-60:i].values.reshape(-1, 1))
       labels.append(y_train.iloc[i])
   features_set, labels = np.array(features_set), np.array(labels)
   features_set = np.reshape(features_set, (features_set.shape[0], features_set.shape[1], 1))
   ```

Build Model

We construct the LSTM model.

1. Build a `Sequential` keras model.

   ```
   model = Sequential()
   ```

2. Create the model infrastructure.

   ```
   # Add our first LSTM layer - 50 nodes.
   model.add(LSTM(units = 50, return_sequences=True, input_shape=(features_set.shape[1], 1)))
   # Add Dropout layer to avoid overfitting
   model.add(Dropout(0.2))
   # Add additional layers
   ```

```
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units = 5))
model.add(Dense(units = 1))
```

3. Compile the model.

   We use Adam as optimizer for adpative step size and MSE as loss function since it is continuous data.

   ```
   model.compile(optimizer = 'adam', loss = 'mean_squared_error', metrics=['mae', 'acc'])
   ```

4. Set early stopping callback method.

   ```
   callback = EarlyStopping(monitor='loss', patience=3, verbose=1, restore_best_weights=True)
   ```

5. Display the model structure.

   ```
   model.summary()
   ```

6. Fit the model to our data, running 20 training epochs.

   Note that different training session's results will not be the same since the batch is randomly selected.

   ```
   model.fit(features_set, labels, epochs = 20, batch_size = 100, callbacks=[callback])
   ```

Test Hypothesis

We would test the performance of this ML model to see if it could predict 1-step forward price precisely. To do so, we would compare the predicted and actual prices.

1. Get testing set features for input.

   ```
   test_features = []
   for i in range(60, X_test.shape[0]):
       test_features.append(X_test.iloc[i-60:i].values.reshape(-1, 1))
   test_features = np.array(test_features)
   test_features = np.reshape(test_features, (test_features.shape[0], test_features.shape[1], 1))
   ```

2. Make predictions.

   ```
   predictions = model.predict(test_features)
   ```

3. Transform predictions back to original data-scale.

   ```
   predictions = scaler.inverse_transform(predictions)
   actual = scaler.inverse_transform(y_test.values)
   ```

4. Plot the results.

   ```
   plt.figure(figsize=(15, 10))
   plt.plot(actual[60:], color='blue', label='Actual')
   plt.plot(predictions , color='red', label='Prediction')
   plt.title('Price vs Predicted Price ')
   plt.legend()
   plt.show()
   ```

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into backtest is to create a scheduled event which uses our model to predict the expected return. If we predict the price will go up, we long SPY, else, we short it.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2016, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.asset = "SPY"

    # Add Equity ------------------------------------------------
    self.AddEquity(self.asset, Resolution.Minute)

    # Initialize the LSTM model
    self.BuildModel()

    # Set Scheduled Event Method For Our Model
    self.Schedule.On(self.DateRules.EveryDay(),
        self.TimeRules.BeforeMarketClose("SPY", 5),
        self.EveryDayBeforeMarketClose)

    # Set Scheduled Event Method For Our Model Retraining every month
    self.Schedule.On(self.DateRules.MonthStart(),
        self.TimeRules.At(0, 0),
        self.BuildModel)
```

We'll also need to create a function to train and update our model from time to time.

```
def BuildModel(self) -> None:
    qb = self

    ### Preparing Data
    # Get historical data
    history = qb.History(qb.Securities.Keys, 252*2, Resolution.Daily)

    # Select the close column and then call the unstack method.
    close = history['close'].unstack(level=0)

    # Scale data onto [0,1]
    self.scaler = MinMaxScaler(feature_range = (0, 1))

    # Transform our data
    df = pd.DataFrame(self.scaler.fit_transform(close), index=close.index)

    # Feature engineer the data for input.
    input_ = df.iloc[1:]

    # Shift the data for 1-step backward as training output result.
    output = df.shift(-1).iloc[:-1]

    # Build feauture and label sets (using number of steps 60, and feature rank 1)
    features_set = []
    labels = []
    for i in range(60, input_.shape[0]):
        features_set.append(input_.iloc[i-60:i].values.reshape(-1, 1))
```

```
        labels.append(output.iloc[i])
    features_set, labels = np.array(features_set), np.array(labels)
    features_set = np.reshape(features_set, (features_set.shape[0], features_set.shape[1], 1))

    ### Build Model
    # Build a Sequential keras model
    self.model = Sequential()

    # Add our first LSTM layer - 50 nodes
    self.model.add(LSTM(units = 50, return_sequences=True, input_shape=(features_set.shape[1], 1)))
    # Add Dropout layer to avoid overfitting
    self.model.add(Dropout(0.2))
    # Add additional layers
    self.model.add(LSTM(units=50, return_sequences=True))
    self.model.add(Dropout(0.2))
    self.model.add(LSTM(units=50))
    self.model.add(Dropout(0.2))
    self.model.add(Dense(units = 5))
    self.model.add(Dense(units = 1))

    # Compile the model. We use Adam as optimizer for adpative step size and MSE as loss function since it is continuous data.
    self.model.compile(optimizer = 'adam', loss = 'mean_squared_error', metrics=['mae', 'acc'])

    # Set early stopping callback method
    callback = EarlyStopping(monitor='loss', patience=3, restore_best_weights=True)

    # Fit the model to our data, running 20 training epochs
    self.model.fit(features_set, labels, epochs = 20, batch_size = 1000, callbacks=[callback])
```

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
def EveryDayBeforeMarketClose(self) -> None:
    qb = self
    # Fetch history on our universe
    history = qb.History(qb.Securities.Keys, 60, Resolution.Daily)
    if history.empty: return

    # Make all of them into a single time index.
    close = history.close.unstack(level=0)

    # Scale our data
    df = pd.DataFrame(self.scaler.transform(close), index=close.index)

    # Feature engineer the data for input
    input_ = []
    input_.append(df.values.reshape(-1, 1))
    input_ = np.array(input_)
    input_ = np.reshape(input_, (input_.shape[0], input_.shape[1], 1))

    # Prediction
    prediction = self.model.predict(input_)

    # Revert the scaling into price
    prediction = self.scaler.inverse_transform(prediction)

    # ==============================

    if prediction > qb.Securities[self.asset].Price:
        self.SetHoldings(self.asset, 1.)
    else:
        self.SetHoldings(self.asset, -1.)
```

Clone Example Project

10.9 Airline Buybacks

Introduction

This page explains how to you can use the Research Environment to develop and test a Airline Buybacks hypothesis, then put the hypothesis in production.

Create Hypothesis

Buyback represents a company buy back its own stocks in the market, as (1) management is confident on its own future, and (2) wants more control over its development. Since usually buyback is in large scale on a schedule, the price of repurchasing often causes price fluctuation.

Airlines is one of the largest buyback sectors. Major US Airlines use over 90% of their free cashflow to buy back their own stocks in the recent years. [1] Therefore, we can use airline companies to test the hypothesis of buybacks would cause price action. In this particular exmaple, we're hypothesizing that difference in buyback price and close price would suggest price change in certain direction. (we don't know forward return would be in momentum or mean-reversion in this case!)

Import Libraries

We'll need to import libraries to help with data processing, validation and visualization. Import `SmartInsiderTransaction` class, `statsmodels` , `sklearn` , `numpy` , `pandas` and `seaborn` libraries by the following:

```
from QuantConnect.DataSource import SmartInsiderTransaction

from statsmodels.discrete.discrete_model import Logit
from sklearn.metrics import confusion_matrix
import numpy as np
import pandas as pd
import seaborn as sns
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Instantiate a `QuantBook` .

   ```
   qb = QuantBook()
   ```

2. Select the airline tickers for research.

   ```
   assets = ["LUV",   # Southwest Airlines
             "DAL",   # Delta Airlines
             "UAL",   # United Airlines Holdings
             "AAL",   # American Airlines Group
             "SKYW",  # SkyWest Inc.
             "ALGT",  # Allegiant Travel Co.
             "ALK"    # Alaska Air Group Inc.
            ]
   ```

3. Call the `AddEquity` method with the tickers, and its corresponding resolution. Then call `AddData` with `SmartInsiderTransaction` to subscribe to their buyback transaction data. Save the `Symbol` s into a dictionary.

   ```
   symbols = {}
   for ticker in assets:
       symbol = qb.AddEquity(ticker, Resolution.Minute).Symbol
       symbols[symbol] = qb.AddData(SmartInsiderTransaction, symbol).Symbol
   ```

   If you do not pass a resolution argument, `Resolution.Minute` is used by default.

4. Call the `History` method with a list of `Symbol` s for all tickers, time argument(s), and resolution to request historical data for the symbols.

   ```
   history = qb.History(list(symbols.keys()), datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

5. Call SPY history as reference.

   ```
   spy = qb.History(qb.AddEquity("SPY").Symbol, datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

6. Call the `History` method with a list of `SmartInsiderTransaction Symbol` s for all tickers, time argument(s), and resolution to request historical data for the symbols.

   ```
   history_buybacks = qb.History(list(symbols.values()), datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
   ```

Prepare Data

We'll have to process our data to get the buyback premium/discount% vs forward return data.

1. Select the close column and then call the `unstack` method.

   ```
   df = history['close'].unstack(level=0)
   spy_close = spy['close'].unstack(level=0)
   ```

2. Call `pct_change` to get the daily return of close price, then shift 1-step backward as prediction.

   ```
   ret = df.pct_change().shift(-1).iloc[:-1]
   ret_spy = spy_close.pct_change().shift(-1).iloc[:-1]
   ```

3. Get the active forward return.

   ```
   active_ret = ret.sub(ret_spy.values, axis=0)
   ```

4. Select the ExecutionPrice column and then call the `unstack` method to get the buyback dataframe.

   ```
   df_buybacks = history_buybacks['executionprice'].unstack(level=0)
   ```

5. Convert buyback history into daily mean data.

   ```
   df_buybacks = df_buybacks.groupby(df_buybacks.index.date).mean()
   df_buybacks.columns = df.columns
   ```

6. Get the buyback premium/discount %.

   ```
   df_close = df.reindex(df_buybacks.index)[~df_buybacks.isna()]
   df_buybacks = (df_buybacks - df_close)/df_close
   ```

7. Create a `Dataframe` to hold the buyback and 1-day forward return data.

   ```
   data = pd.DataFrame(columns=["Buybacks", "Return"])
   ```

8. Append the data into the `Dataframe` .

   ```
   for row, row_buyback in zip(active_ret.reindex(df_buybacks.index).itertuples(), df_buybacks.itertuples()):
       index = row[0]
       for i in range(1, df_buybacks.shape[1]+1):
           if row_buyback[i] != 0:
               data = pd.concat([data, pd.DataFrame({"Buybacks": row_buyback[i], "Return":row[i]}, index=[index])])
   ```

9. Call `dropna` to drop NaNs.

   ```
   data.dropna(inplace=True)
   ```

Test Hypothesis

We would test (1) if buyback has statistically significant effect on return direction, and (2) buyback could be a return predictor.

1. Get binary return (+/-).

```
binary_ret = data["Return"].copy()
binary_ret[binary_ret < 0] = 0
binary_ret[binary_ret > 0] = 1
```

2. Construct a logistic regression model.

```
model = Logit(binary_ret.values, data["Buybacks"].values).fit()
```

3. Display logistic regression results.

```
display(model.summary())
```

We can see a p-value of < 0.05 in the logistic regression model, meaning the separation of positive and negative using buyback premium/discount% is statistically significant.

4. Plot the results.

```
plt.figure(figsize=(10, 6))
sns.regplot(x=data["Buybacks"]*100, y=binary_ret, logistic=True, ci=None, line_kws={'label': " Logistic Regression Line"})
plt.plot([-50, 50], [0.5, 0.5], "r--", label="Selection Cutoff Line")
plt.title("Buyback premium vs Profit/Loss")
plt.xlabel("Buyback premium %")
plt.xlim([-50, 50])
plt.ylabel("Profit/Loss")
plt.legend()
plt.show()
```

Interesting, from the logistic regression line, we observe that when the airlines brought their stock in premium price, the price tended to go down, while the opposite for buying back in discount.

Let's also study how good is the logistic regression.

5. Get in-sample prediction result.

```
predictions = model.predict(data["Buybacks"].values)
for i in range(len(predictions)):
    predictions[i] = 1 if predictions[i] > 0.5 else 0
```

6. Call `confusion_matrix` to contrast the results.

```
cm = confusion_matrix(binary_ret, predictions)
```

7. Display the result.

```
df_result = pd.DataFrame(cm,
                        index=pd.MultiIndex.from_tuples([("Prediction", "Positive"), ("Prediction", "Negative")]),
                        columns=pd.MultiIndex.from_tuples([("Actual", "Positive"), ("Actual", "Negative")]))
```

The logistic regression is having a 55.8% accuracy (55% sensitivity and 56.3% specificity), this can suggest a > 50% win rate before friction costs, proven our hypothesis.

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting. One way to accomodate this model into backtest is to create a scheduled event which uses our model to predict the expected return.

```
def Initialize(self) -> None:

    #1. Required: Five years of backtest history
    self.SetStartDate(2017, 1, 1)

    #2. Required: Alpha Streams Models:
    self.SetBrokerageModel(BrokerageName.AlphaStreams)

    #3. Required: Significant AUM Capacity
    self.SetCash(1000000)

    #4. Required: Benchmark to SPY
    self.SetBenchmark("SPY")

    self.SetPortfolioConstruction(EqualWeightingPortfolioConstructionModel())
    self.SetExecution(ImmediateExecutionModel())

    # Set our strategy to be take 5% profit and 5% stop loss.
    self.AddRiskManagement(MaximumUnrealizedProfitPercentPerSecurity(0.05))
    self.AddRiskManagement(MaximumDrawdownPercentPerSecurity(0.05))

    # Select the airline tickers for research.
    self.symbols = {}
    assets = ["LUV",   # Southwest Airlines
              "DAL",   # Delta Airlines
              "UAL",   # United Airlines Holdings
              "AAL",   # American Airlines Group
              "SKYW",  # SkyWest Inc.
              "ALGT",  # Allegiant Travel Co.
              "ALK"    # Alaska Air Group Inc.
              ]

    # Call the AddEquity method with the tickers, and its corresponding resolution. Then call AddData with SmartInsiderTransaction to subscribe to their buyback transaction data.
    for ticker in assets:
        symbol = self.AddEquity(ticker, Resolution.Minute).Symbol
        self.symbols[symbol] = self.AddData(SmartInsiderTransaction, symbol).Symbol

    self.AddEquity("SPY")

    # Initialize the model
    self.BuildModel()

    # Set Scheduled Event Method For Our Model Recalibration every month
    self.Schedule.On(self.DateRules.MonthStart(), self.TimeRules.At(0, 0), self.BuildModel)

    # Set Scheduled Event Method For Trading
    self.Schedule.On(self.DateRules.EveryDay(), self.TimeRules.BeforeMarketClose("SPY", 5), self.EveryDayBeforeMarketClose)
```

We'll also need to create a function to train and update the logistic regression model from time to time.

```
def BuildModel(self) -> None:
    qb = self
    # Call the History method with list of tickers, time argument(s), and resolution to request historical data for the symbol.
    history = qb.History(list(self.symbols.keys()), datetime(2015, 1, 1), datetime.now(), Resolution.Daily)

    # Call SPY history as reference
    spy = qb.History(["SPY"], datetime(2015, 1, 1), datetime.now(), Resolution.Daily)

    # Call the History method with list of buyback tickers, time argument(s), and resolution to request buyback data for the symbol.
    history_buybacks = qb.History(list(self.symbols.values()), datetime(2015, 1, 1), datetime.now(), Resolution.Daily)

    # Select the close column and then call the unstack method to get the close price dataframe.
    df = history['close'].unstack(level=0)
    spy_close = spy['close'].unstack(level=0)

    # Call pct_change to get the daily return of close price, then shift 1-step backward as prediction.
    ret = df.pct_change().shift(-1).iloc[:-1]
    ret_spy = spy_close.pct_change().shift(-1).iloc[:-1]
```

```
    # Get the active return
    active_ret = ret.sub(ret_spy.values, axis=0)

    # Select the ExecutionPrice column and then call the unstack method to get the dataframe.
    df_buybacks = history_buybacks['executionprice'].unstack(level=0)

    # Convert buyback history into daily mean data
    df_buybacks = df_buybacks.groupby(df_buybacks.index.date).mean()
    df_buybacks.columns = df.columns

    # Get the buyback premium/discount
    df_close = df.reindex(df_buybacks.index)[~df_buybacks.isna()]
    df_buybacks = (df_buybacks - df_close)/df_close

    # Create a dataframe to hold the buyback and 1-day forward return data
    data = pd.DataFrame(columns=["Buybacks", "Return"])

    # Append the data into the dataframe
    for row, row_buyback in zip(active_ret.reindex(df_buybacks.index).itertuples(), df_buybacks.itertuples()):
        index = row[0]
        for i in range(1, df_buybacks.shape[1]+1):
            if row_buyback[i] != 0:
                data = pd.concat([data, pd.DataFrame({"Buybacks": row_buyback[i], "Return":row[i]}, index=[index])])

    # Call dropna to drop NaNs
    data.dropna(inplace=True)

    # Get binary return (+/-)
    binary_ret = data["Return"].copy()
    binary_ret[binary_ret < 0] = 0
    binary_ret[binary_ret > 0] = 1

    # Construct a logistic regression model
    self.model = Logit(binary_ret.values, data["Buybacks"].values).fit()
```

Now we export our model into the scheduled event method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
def EveryDayBeforeMarketClose(self) -> None:
    qb = self
    # Get any buyback event today
    history_buybacks = qb.History(list(self.symbols.values()), timedelta(days=1), Resolution.Daily)
    if history_buybacks.empty or "executionprice" not in history_buybacks.columns: return

    # Select the ExecutionPrice column and then call the unstack method to get the dataframe.
    df_buybacks = history_buybacks['executionprice'].unstack(level=0)

    # Convert buyback history into daily mean data
    df_buybacks = df_buybacks.groupby(df_buybacks.index.date).mean()

    # ==============================

    insights = []

    # Iterate the buyback data, thne pass to the model for prediction
    row = df_buybacks.iloc[-1]
    for i in range(len(row)):
        prediction = self.model.predict(row[i])

        # Long if the prediction predict price goes up, short otherwise. Do opposite for SPY (active return)
        if prediction > 0.5:
            insights.append( Insight.Price(row.index[i].split(".")[0], timedelta(days=1), InsightDirection.Up) )
            insights.append( Insight.Price("SPY", timedelta(days=1), InsightDirection.Down) )
        else:
            insights.append( Insight.Price(row.index[i].split(".")[0], timedelta(days=1), InsightDirection.Down) )
            insights.append( Insight.Price("SPY", timedelta(days=1), InsightDirection.Up) )

    self.EmitInsights(insights)
```

Reference

- US Airlines Spent 96% of Free Cash Flow on Buybacks: Chart. B. Kochkodin (17 March 2020). *Bloomberg. Retrieve from: https://www.bloomberg.com/news/articles/2020-03-16/u-s-airlines-spent-96-of-free-cash-flow-on-buybacks-chart.*

Clone Example Project

- Charts
- Statistics
- Code
  - main.py
  - research.ipynb

Clone Algorithm

Overall Statistics	
Total Trades	448
Average Win	0.40%
Average Loss	-0.33%
Compounding Annual Return	0.012%
Drawdown	8.400%
Expectancy	0.010
Net Profit	0.058%
Sharpe Ratio	0.02
Probabilistic Sharpe Ratio	0.399%
Loss Rate	54%
Win Rate	46%
Profit-Loss Ratio	1.20
Alpha	0.001
Beta	-0.004
Annual Standard Deviation	0.036
Annual Variance	0.001
Information Ratio	-0.847
Tracking Error	0.161
Treynor Ratio	-0.168
Total Fees	$8583.04
Estimated Strategy Capacity	$1500000.00
Lowest Capacity Asset	DAL TSAT9G6HOJ8L

```
#region imports
from AlgorithmImports import *
from statsmodels.discrete.discrete_model import Logit
#endregion

class AirlineBuybacksDemo(QCAlgorithm):

    def Initialize(self):
        #1. Required: Five years of backtest history
        self.SetStartDate(2017, 1, 1)
```

10.10 Sparse Optimization

Introduction

This page explains how to you can use the Research Environment to develop and test a Sparse Optimization Index Tracking hypothesis, then put the hypothesis in production.

Create Hypothesis

Passive index fund portfolio managers will buy in corresponding weighting of stocks from an index's constituents. The main idea is allowing market participants to trade an index in a smaller cost. Their performance is measured by Tracking Error (TE), which is the standard deviation of the active return of the portfolio versus its benchmark index. The lower the TE means that the portfolio tracks the index very accurately and consistently.

A technique called Sparse Optimization comes into the screen as the portfolio managers want to cut their cost even lower by trading less frequently and with more liquid stocks. They select a desired group/all constituents from an index and try to strike a balance between the number of stocks in the portfolio and the TE, like the idea of L1/L2-normalization.

On the other hand, long-only active fund aimed to beat the benchmark index. Their performance are measured by the mean-adjusted tracking error, which also take the mean active return into account, so the better fund can be identified as consisitently beating the index by n%.

We can combine the 2 ideas. In this tutorial, we are about to generate our own active fund and try to use Sparse Optimization to beat QQQ. However, we need a new measure on active fund for this technique -- Downward Risk (DR). This is a measure just like the tracking error, but taking out the downward period of the index, i.e. we only want to model the index's upward return, but not downward loss. We would also, for a more robust regression, combining Huber function as our loss function. This is known as Huber Downward Risk (HDR). Please refer to [Optimization Methods for Financial Index Tracking: From Theory to Practice. *K. Benidis, Y. Feng, D. P. Palomer (2018)*] for technical details.

Import Libraries

We'll need to import libraries to help with data processing and visualization. Import `numpy` , `matplotlib` and `pandas` libraries by the following:

```
import numpy as np
from matplotlib import pyplot as plt
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Get Historical Data

To begin, we retrieve historical data for researching.

1. Create a class to get the index/ETF constituents on a particular date.

```
class ETFUniverse:
    """
    A class to create a universe of equities from the constituents of an ETF
    """
    def __init__(self, etf_ticker, universe_date):
        """
        Input:
         - etf_ticker
            Ticker of the ETF
         - universe_date
            The date to gather the constituents of the ETF
        """
        self.etf_ticker = etf_ticker
        self.universe_date = universe_date


    def get_symbols(self, qb):
        """
        Subscribes to the universe constituents and returns a list of symbols and their timezone

        Input:
         - qb
            The QuantBook instance inside the DatasetAnalyzer

        Returns a list of symbols and their timezone
        """
        etf_symbols = self._get_etf_constituents(qb, self.etf_ticker, self.universe_date)
        security_timezone = None
        security_symbols = []

        # Subscribe to the universe price data
        for symbol in etf_symbols:
            security = qb.AddSecurity(symbol, Resolution.Daily)
            security_timezone = security.Exchange.TimeZone
            security_symbols.append(symbol)

        return security_symbols, security_timezone


    def _get_etf_constituents(self, qb, etf_ticker, date):
        """
        A helper method to retreive the ETF constituents on a given date

        Input:
         - qb
            The QuantBook instance inside the DatasetAnalyzer
         - etf_ticker
             Ticker of the ETF
         - universe_date
            The date to gather the constituents of the ETF

        Returns a list of symbols
        """
        date_str = date.strftime("%Y%m%d")
        filename = f"/data/equity/usa/universes/etf/{etf_ticker.lower()}/{date_str}.csv"
        try:
            df = pd.read_csv(filename)
        except:
            print(f'Error: The ETF universe file does not exist')
            return
        security_ids = df[df.columns[1]].values
        symbols = [qb.Symbol(security_id) for security_id in security_ids]
        return symbols
```

2. Instantiate a `QuantBook` .

```
qb = QuantBook()
```

3. Subscribe to the index/ETF.

   In this tutorial, we'll be using QQQ.

```
qqq = qb.AddEquity("QQQ").Symbol
```

4. Select all the constituents for research.

   In this tutorial, we select the constituents of QQQ on 2020-12-31.

```
assets, _ = ETFUniverse("QQQ", datetime(2020, 12, 31)).get_symbols(qb)
```

5. Prepare the historical return data of the constituents and the benchmark index to track.

```
spy = qb.History(qb.AddEquity("SPY").Symbol, datetime(2019, 1, 1), datetime(2021, 12, 31), Resolution.Daily)
```

6. Call the `History` method with a list of `SmartInsiderTransaction Symbol` s for all tickers, time argument(s), and resolution to request historical data for the symbols.

```
history = qb.History(assets, datetime(2020, 1, 1), datetime(2021, 3, 31), Resolution.Daily)
```

```
historyPortfolio = history.close.unstack(0).loc[:"2021-01-01"]
pctChangePortfolio = np.log(historyPortfolio/historyPortfolio.shift(1)).dropna()

historyQQQ_ = qb.History(qqq, datetime(2020, 1, 1), datetime(2021, 3, 31), Resolution.Daily)
historyQQQ = historyQQQ_.close.unstack(0).loc[:"2021-01-01"]
pctChangeQQQ = np.log(historyQQQ/historyQQQ.shift(1)).loc[pctChangePortfolio.index]
```

Prepare Data

We'll have to process our data and construct the proposed sparse index tracking portfolio.

1. Get the dimensional sizes.

```
m = pctChangePortfolio.shape[0]; n = pctChangePortfolio.shape[1]
```

2. Set up optimization parameters (penalty of exceeding bounds, Huber statistics M-value, penalty weight).

```
p = 0.5
M = 0.0001
l = 0.01
```

3. Set up convergence tolerance, maximum iteration of optimization, iteration counter and HDR as minimization indicator.

```
tol = 0.001
maxIter = 20
iters = 1
hdr = 10000
```

4. Initial weightings and placeholders.

```
w_ = np.array([1/n] * n).reshape(n, 1)
weights = pd.Series()
a = np.array([None] * m).reshape(m, 1)
c = np.array([None] * m).reshape(m, 1)
d = np.array([None] * n).reshape(n, 1)
```

5. Iterate minimization algorithm to minimize the HDR.

```
while iters < maxIter:
    x_k = (pctChangeQQQ.values - pctChangePortfolio.values @ w_)
    for i in range(n):
        w = w_[i]
        d[i] = d_ = 1/(np.log(1+l/p)*(p+w))
    for i in range(m):
        xk = float(x_k[i])
        if xk < 0:
            a[i] = M / (M - 2*xk)
            c[i] = xk
        else:
            c[i] = 0
            if 0 <= xk <= M:
                a[i] = 1
            else:
                a[i] = M/abs(xk)

    L3 = 1/m * pctChangePortfolio.T.values @ np.diagflat(a.T) @ pctChangePortfolio.values
    eigVal, eigVec = np.linalg.eig(L3.astype(float))
    eigVal = np.real(eigVal); eigVec = np.real(eigVec)
    q3 = 1/max(eigVal) * (2 * (L3 - max(eigVal) * np.eye(n)) @ w_ + eigVec @ d - 2/m * pctChangePortfolio.T.values @ np.diagflat(a.T) @ (c - pctChangeQQQ.values))

    # We want to keep the upper bound of each asset to be 0.1
    u = 0.1
    mu = float(-(np.sum(q3) + 2)/n); mu_ = 0
    while mu > mu_:
        mu = mu_
        index1 = [i for i, q in enumerate(q3) if mu + q < -u*2]
        index2 = [i for i, q in enumerate(q3) if -u*2 < mu + q < 0]
        mu_ = float(-(np.sum([q3[i] for i in index2]) + 2 - len(index1)*u*2)/len(index2))

    # Obtain the weights and HDR of this iteration.
    w_ = np.amax(np.concatenate((-(mu + q3)/2, u*np.ones((n, 1))), axis=1), axis=1).reshape(-1, 1)
    w_ = w_/np.sum(abs(w_))
    hdr_ = float(w_.T @ w_ + q3.T @ w_)

    # If the HDR converges, we take the current weights
    if abs(hdr - hdr_) < tol:
        break

    # Else, we would increase the iteration count and use the current weights for the next iteration.
    iters += 1
    hdr = hdr_
```

6. Save the final weights.

```
for i in range(n):
    weights[pctChangePortfolio.columns[i]] = w_[i]
```

7. Get the historical return of the proposed portfolio.

```
histPort = historyPortfolio.dropna() @ np.array([weights[pctChangePortfolio.columns[i]] for i in range(pctChangePortfolio.shape[1])])
```

Test Hypothesis

To test the hypothesis. We wish to (1) outcompete the benchmark and (2) the active return is consistent in the in- and out-of-sample period.

1. Obtain the equity curve of our portfolio and normalized benchmark for comparison.

```
proposed = history.close.unstack(0).dropna() @ np.array([weights[pctChangePortfolio.columns[i]] for i in range(pctChangePortfolio.shape[1])])
benchmark = historyQQQ_.close.unstack(0).loc[proposed.index]
normalized_benchmark = benchmark / (float(benchmark.iloc[0])/float(proposed.iloc[0]))
```

2. Obtain the active return.

```
proposed_ret = proposed.pct_change().iloc[1:]
benchmark_ret = benchmark.pct_change().iloc[1:]
active_ret = proposed_ret - benchmark_ret.values
```

3. Plot the result.

```
fig = plt.figure(figsize=(15, 10))
plt.plot(proposed, label="Proposed Portfolio")
plt.plot(normalized_benchmark, label="Normalized Benchmark")
min_ = min(min(proposed.values), min(normalized_benchmark.values))
max_ = max(max(proposed.values), max(normalized_benchmark.values))
plt.plot([pd.to_datetime("2021-01-01")]*100, np.linspace(min_, max_, 100), "r--", label="in- and out- of sample separation")
plt.title("Equity Curve")
plt.legend()
plt.show()
plt.clf()

fig, ax = plt.subplots(1, 1)
active_ret["Mean"] = float(active_ret.mean())
active_ret.plot(figsize=(15, 5), title="Active Return", ax=ax)
plt.show()
```

We can see from the plots, both in- and out-of-sample period the proposed portfolio out preform the benchmark while remaining a high correlation with it. Although the active return might not be very consistent, but it is a stationary series above zero. So, in a long run, it does consistently outcompete the QQQ benchmark!

Set Up Algorithm

Once we are confident in our hypothesis, we can export this code into backtesting.

```
def Initialize(self) -> None:
    self.SetStartDate(2017, 1, 1)
    self.SetBrokerageModel(BrokerageName.AlphaStreams)
    self.SetCash(1000000)

    # Add our ETF constituents of the index that we would like to track.
    self.QQQ = self.AddEquity("QQQ", Resolution.Minute).Symbol
    self.UniverseSettings.Resolution = Resolution.Minute
    self.AddUniverse(self.Universe.ETF(self.QQQ, self.UniverseSettings, self.ETFSelection))

    self.SetBenchmark("QQQ")

    # Set up varaibles to flag the time to recalibrate and hold the constituents.
    self.time = datetime.min
    self.assets = []
```

We'll also need to create a function for getting the ETF constituents.

```
def ETFSelection(self, constituents: ETFConstituentData) -> List[Symbol]:
    # We want all constituents to be considered.
    self.assets = [x.Symbol for x in constituents]
    return self.assets
```

Now we export our model into the `OnData` method. We will switch `qb` with `self` and replace methods with their `QCAlgorithm` counterparts as needed. In this example, this is not an issue because all the methods we used in research also exist in `QCAlgorithm` .

```
def OnData(self, slice: Slice) -> None:
    qb = self
    if self.time > self.Time:
        return

    # Prepare the historical return data of the constituents and the ETF (as index to track).
    history = qb.History(self.assets, 252, Resolution.Daily)
    if history.empty: return

    historyPortfolio = history.close.unstack(0)
    pctChangePortfolio = np.log(historyPortfolio/historyPortfolio.shift(1)).dropna()

    historyQQQ = qb.History(self.AddEquity("QQQ").Symbol, 252, Resolution.Daily)
    historyQQQ = historyQQQ.close.unstack(0)
    pctChangeQQQ = np.log(historyQQQ/historyQQQ.shift(1)).loc[pctChangePortfolio.index]

    m = pctChangePortfolio.shape[0]; n = pctChangePortfolio.shape[1]

    # Set up optimization parameters.
    p = 0.5; M = 0.0001; l = 0.01

    # Set up convergence tolerance, maximum iteration of optimization, iteration counter and Huber downward risk as minimization indicator.
    tol = 0.001; maxIter = 20; iters = 1; hdr = 10000

    # Initial weightings and placeholders.
    w_ = np.array([1/n] * n).reshape(n, 1)
    self.weights = pd.Series()
    a = np.array([None] * m).reshape(m, 1)
    c = np.array([None] * m).reshape(m, 1)
    d = np.array([None] * n).reshape(n, 1)

    # Iterate to minimize the HDR.
    while iters < maxIter:
        x_k = (pctChangeQQQ.values - pctChangePortfolio.values @ w_)
        for i in range(n):
            w = w_[i]
            d[i] = d_ = 1/(np.log(1+l/p)*(p+w))
        for i in range(m):
            xk = float(x_k[i])
            if xk < 0:
                a[i] = M / (M - 2*xk)
                c[i] = xk
            else:
                c[i] = 0
                if 0 <= xk <= M:
                    a[i] = 1
                else:
                    a[i] = M/abs(xk)

        L3 = 1/m * pctChangePortfolio.T.values @ np.diagflat(a.T) @ pctChangePortfolio.values
        eigVal, eigVec = np.linalg.eig(L3.astype(float))
        eigVal = np.real(eigVal); eigVec = np.real(eigVec)
        q3 = 1/max(eigVal) * (2 * (L3 - max(eigVal) * np.eye(n)) @ w_ + eigVec @ d - 2/m * pctChangePortfolio.T.values @ np.diagflat(a.T) @ (c - pctChangeQQQ.values))

        # We want to keep the upper bound of each asset to be 0.1
        u = 0.1
        mu = float(-(np.sum(q3) + 2)/n); mu_ = 0
        while mu > mu_:
            mu = mu_
            index1 = [i for i, q in enumerate(q3) if mu + q < -u*2]
            index2 = [i for i, q in enumerate(q3) if -u*2 < mu + q < 0]
            mu_ = float(-(np.sum([q3[i] for i in index2]) + 2 - len(index1)*u*2)/len(index2))

        # Obtain the weights and HDR of this iteration.
        w_ = np.amax(np.concatenate((-(mu + q3)/2, u*np.ones((n, 1))), axis=1), axis=1).reshape(-1, 1)
        w_ = w_/np.sum(abs(w_))
        hdr_ = float(w_.T @ w_ + q3.T @ w_)

        # If the HDR converges, we take the current weights
        if abs(hdr - hdr_) < tol:
            break

        # Else, we would increase the iteration count and use the current weights for the next iteration.
        iters += 1
        hdr = hdr_

    # -----------------------------------------------------------------------------------------
    orders = []
    for i in range(n):
        orders.append(PortfolioTarget(pctChangePortfolio.columns[i], float(w_[i])))
    self.SetHoldings(orders)

    # Recalibrate on quarter end.
    self.time = Expiry.EndOfQuarter(self.Time)
```

Reference

- Optimization Methods for Financial Index Tracking: From Theory to Practice. K. Benidis, Y. Feng, D. P. Palomer (2018). *Foundations and Trends in Signal Processing. 3-3. p171-279.*

Clone Example Project

Overall Statistics	
Total Trades	302
Average Win	0.24%
Average Loss	-0.30%
Compounding Annual Return	-15.479%
Drawdown	28.600%
Expectancy	-0.478
Net Profit	-16.704%
Sharpe Ratio	-0.333
Probabilistic Sharpe Ratio	4.606%
Loss Rate	71%
Win Rate	29%
Profit-Loss Ratio	0.79
Alpha	0.047
Beta	0.909
Annual Standard Deviation	0.247
Annual Variance	0.061
Information Ratio	0.771
Tracking Error	0.077
Treynor Ratio	-0.091
Total Fees	$413.22
Estimated Strategy Capacity	$4000.00
Lowest Capacity Asset	QCOM R735QTJ8XC9X

```
#region imports
from AlgorithmImports import *
#endregion

class SparseOptimizationIndexTrackingDemo(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2017, 1, 1)
        self.SetStartDate(2022, 1, 1)
        self.SetBrokerageModel(BrokerageName.AlphaStreams)
```