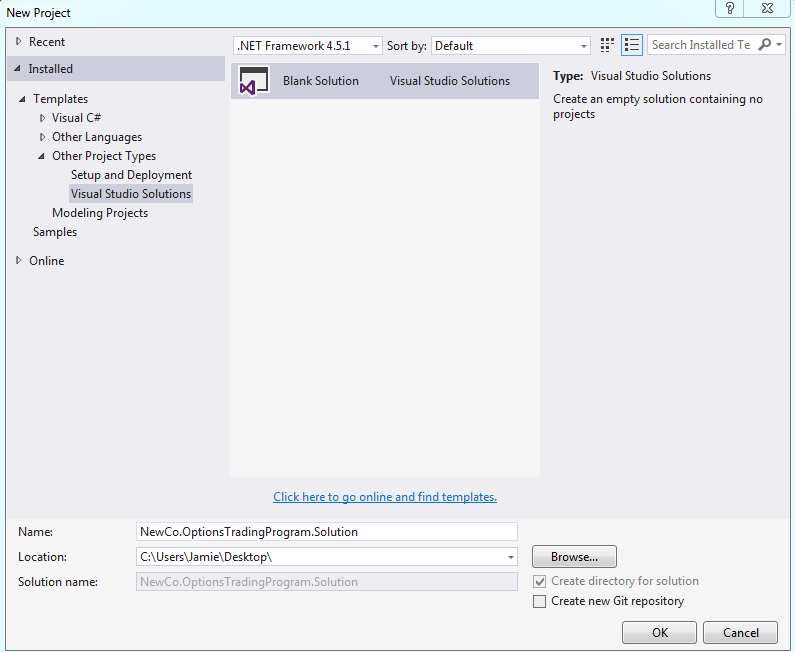
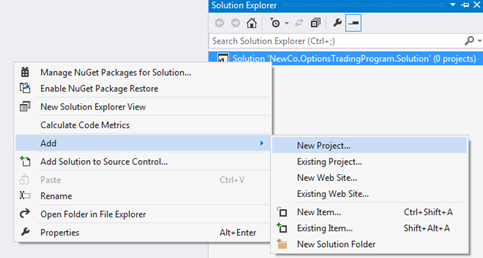
**Set Up The Solution**

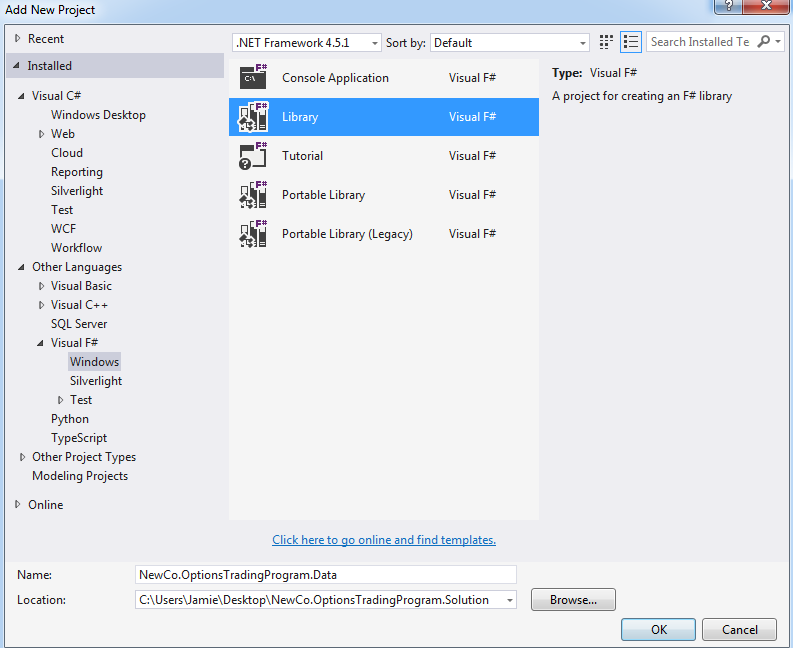
Open Visual Studio

Create a new solution (File -> New -> Project) called NewCo.OptionsTradingProgram.Solution

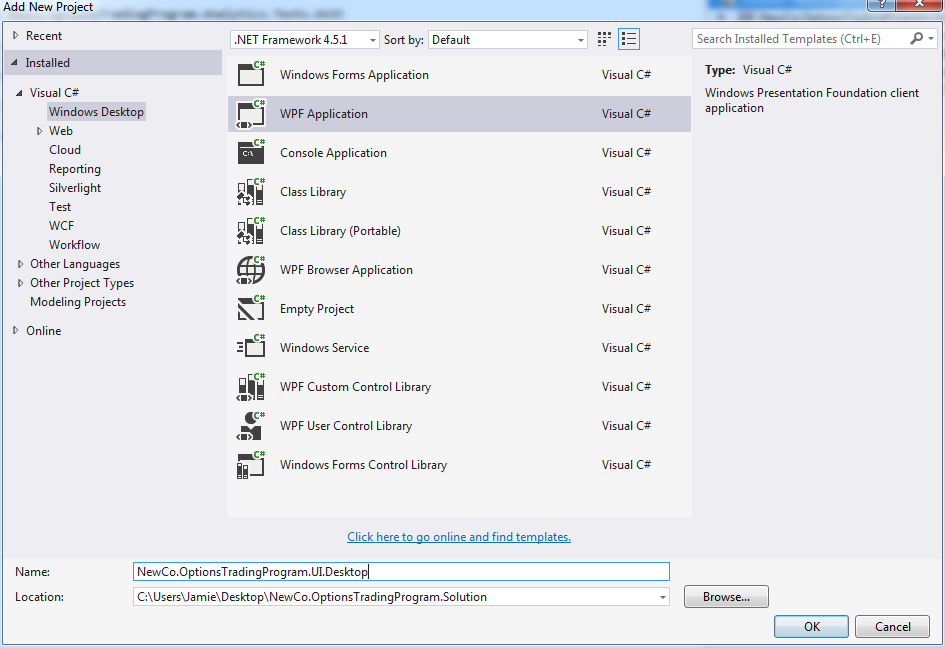


In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual F# -> Windows -> Library project. Call it NewCo.OptionsTradingProgram.Data

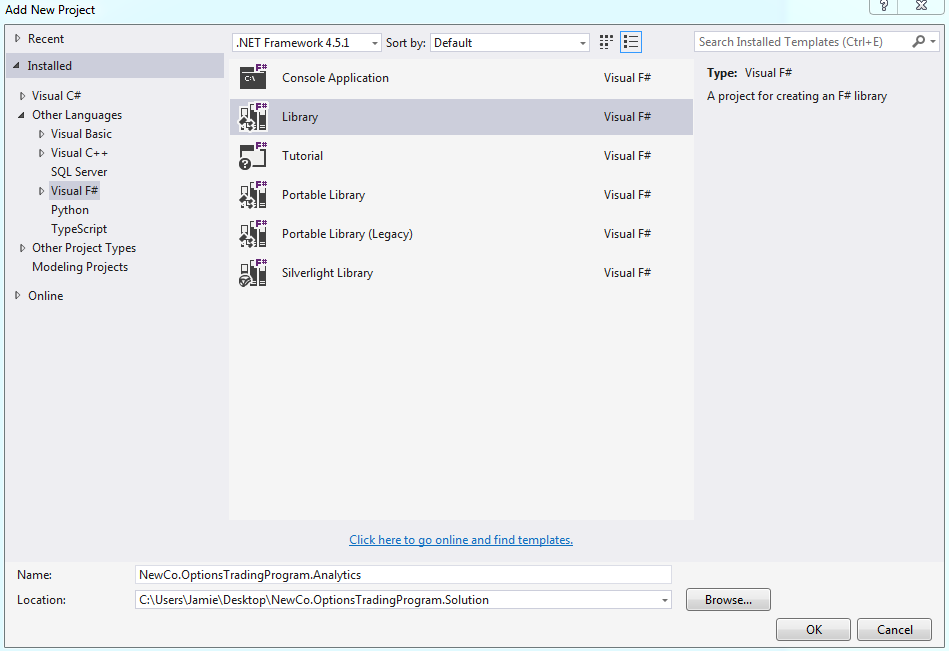




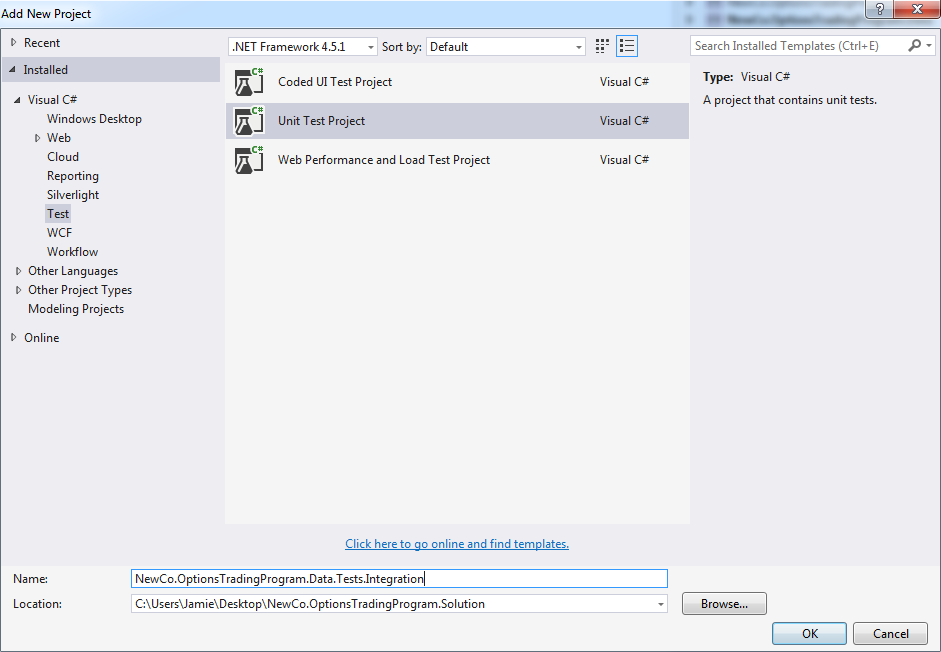
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Windows Desktop -> WPF Application. Call it NewCo.OptionsTradingProgram.UI.Desktop



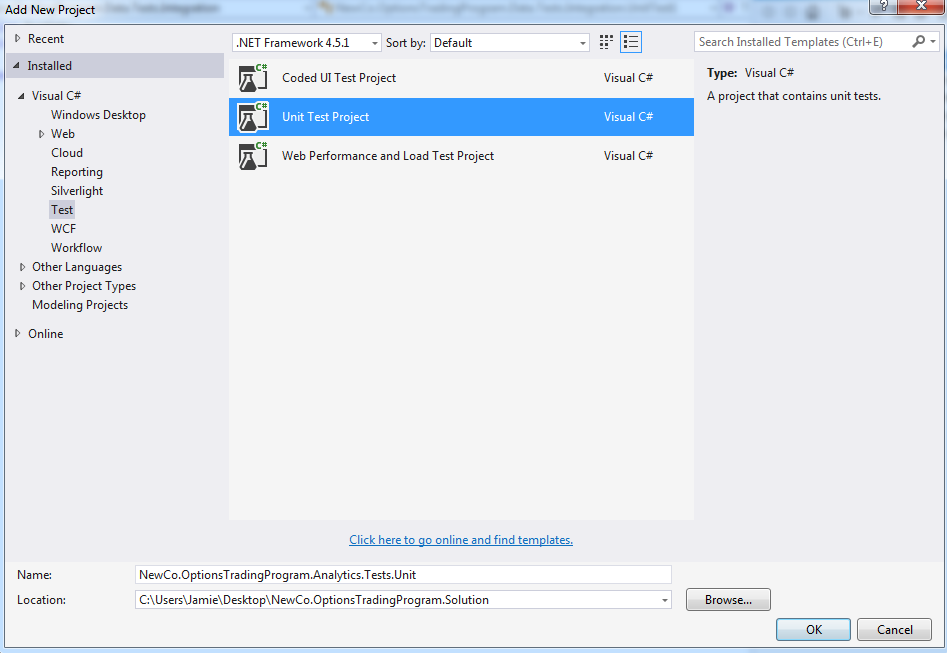
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual F# -> Windows -> Library project. Call it NewCo.OptionsTradingProgram.Analytics



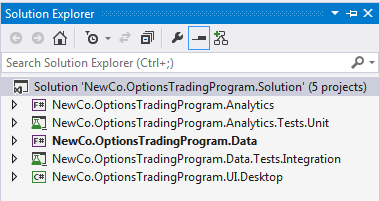
In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Test -> Unit Test Project. Call it NewCo.OptionsTradingProgram.Data.Tests.Integration.



In the solution explorer, add a new project (Solution Explorer -> Right Click -> New Project) and select Visual C# -> Test -> Unit Test Project. Call it NewCo.OptionsTradingProgram.Analytics.Tests.Unit.

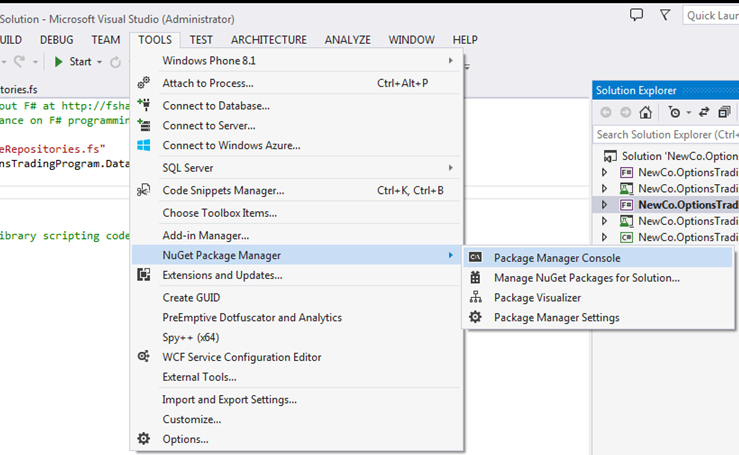


Your solution explorer should look like this

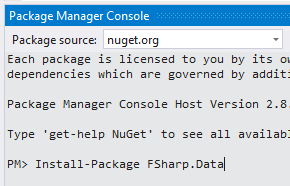


**Access The Data**

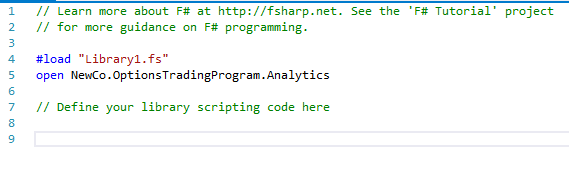
Open the NewCo.OptionsTradingProgram.Data project and open the NuGet package Manager Console (TOOLS -> NuGetPackageManager->Package Manager Console)



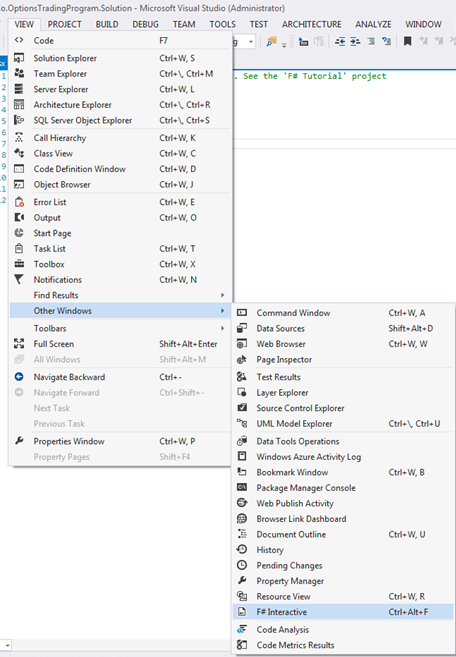
At the command prompt, enter the following command:



Open the script.fsx file.



Open the F# Interactive (the built-in REPL)



We are now ready to “prove out” data access from Yahoo.

In the Script.fsx file, remove all code and type in the following bock.

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

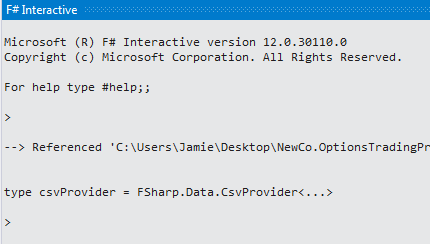
open FSharp.Data

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

Note that the actual path will be different for your machine (wherever Nuget stuck the file). You can open the folder via solution explorer and copy/paste the path so you don’t have to type the path

Once you finish with the #r line, you will get a permission dialog box, Hit “enable”.

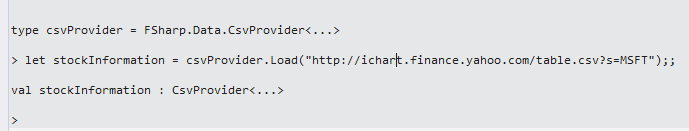
Highlight all 4 lines and hit ALT+ENTER. In your FSI, you should see:



We are going to now inspect the data in the REPL. At the “>”, enter the following line:

let stockInformation = csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=MSFT");;

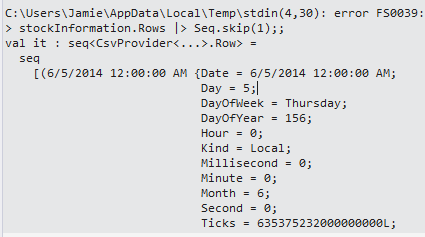
Note that there is a two semi-colons in the REPL. Your REPL should look like this:



Next, type this:

stockInformation.Rows |> Seq.Skip(1);;

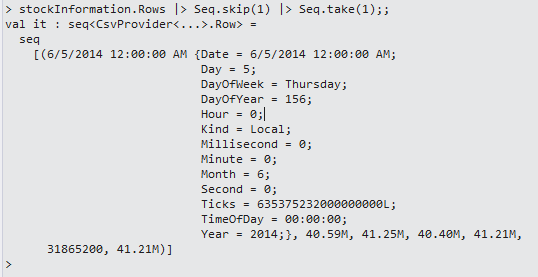
You should get something like this:



Then, type this into the REPL:

stockInformation.Rows |> Seq.skip(1) |> Seq.take(1);;

And you should get something like this:



Which is the 2nd row of data coming back from Yahoo. If you want to see the column headers, type this into the REPL (Note that you can hit the up arrow if you don’t want to type the same commands in again):

stockInformation.Headers;;

And you should get something like this:



With our concept ‘proved out’ in the REPL, we are ready to move the code into an actual class that can be consumed by other projects. Go to Library1.fs in the same project and rename it to StockPriceRepositories.fs. Remove all code and replace it with this:

namespace NewCo.OptionsTradingProgram.Data

open System

open FSharp.Data

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol)

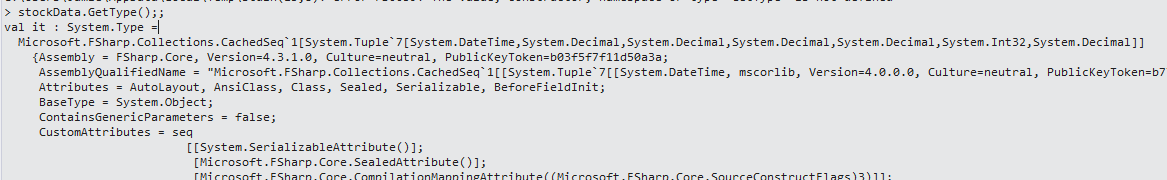
We now have a class that pulls data from Yahoo that we can use in our analysis. As we continue to progress with our project, we will want to pull this data down often. It would be much better to hold a sample of this data locally so that

1. Yahoo doesn’t throttle us
2. We have the same dataset every time for controlled experiments

We could hard-code in a sample dataset into memory, but that is a lot of typing. Instead, let’s pull the data down one time and store it on disk. That way, we can pull it up when needed. The barrier is that the type of the csvProvider is inferred. You can see this by entering this into the REPL:

stockData.GetType();;

Which returns:

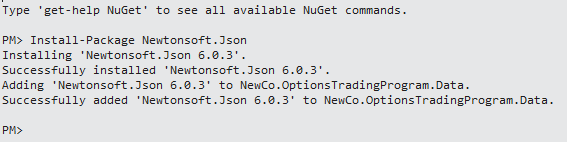


So once we start using different providers of the same type, we should define the type and then have a way of serializing and deserializing it. Let’s add Json.Net to our project so we can serialize and deserialize to Json.

Open the Package Manager Console and type this

Install-Package Newtonsoft.Json

You should get this



With Newtonsoft.Json installed, we can reference it in the REPL.

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\Newtonsoft.Json.6.0.3\lib\\net45\Newtonsoft.Json.dll"

open System

open FSharp.Data

open Newtonsoft.Json

Next, add this below Newtonsoft.Json

open System.IO

At the bottom of the script, add these lines:

type FileSystemStockProvider() =

member this.PutData(filePath:string, stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

member this.GetData(filePath:string) =

let serializedData = File.ReadAllLines(filePath)

serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

The entire script should look like this:

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\FSharp.Data.2.0.8\lib\portable-net40+sl5+wp8+win8\FSharp.Data.dll"

#r "C:\Users\Jamie\Desktop\NewCo.OptionsTradingProgram.Solution\packages\Newtonsoft.Json.6.0.3\lib\\net45\Newtonsoft.Json.dll"

open System

open FSharp.Data

open Newtonsoft.Json

open System.IO

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol).Rows

type FileSystemStockProvider() =

member this.PutData(filePath:string, stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

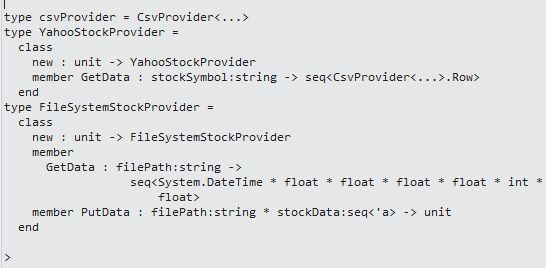
member this.GetData(filePath:string) =

let serializedData = File.ReadAllLines(filePath)

serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

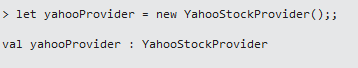
Highlight everything and hit ALT+ENTER. The REPL should look like:



In the REPL, enter this at the >

let yahooProvider = new YahooStockProvider();;

And you should get this



Next, enter this:

let yahooData = yahooProvider.GetData("MSFT");;

And you should get this:



Next, enter this:

let fileSystemProvider = new FileSystemStockProvider();;

And you should get this:



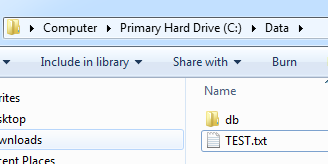
Next, enter this:

fileSystemProvider.PutData("C:\Data\TEST.txt",yahooData);;

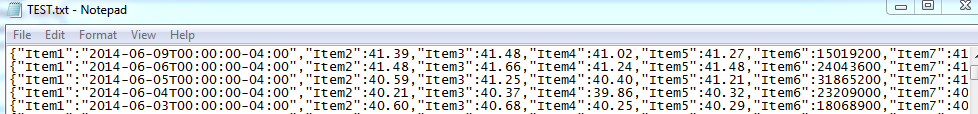
And you should get this:



And if you open your file system, you should see the file:



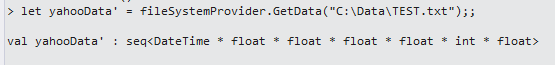
And if you open the file, you can see the contents:



Back to the REPL, enter this:

let yahooData' = fileSystemProvider.GetData("C:\Data\TEST.txt");;

And you should get this:



So now we have a way of getting data from Yahoo and read and writing it to the file system. Let’s take the code from the .fsx file and put it into the StockPriceRepositories file:

namespace NewCo.OptionsTradingProgram.Data

open System

open FSharp.Data

open Newtonsoft.Json

open System.IO

type csvProvider = CsvProvider<"http://ichart.finance.yahoo.com/table.csv?s=MSFT">

type YahooStockProvider() =

member this.GetData(stockSymbol: string) =

csvProvider.Load("http://ichart.finance.yahoo.com/table.csv?s=" + stockSymbol).Rows

type FileSystemStockProvider(filePath:string) =

member this.PutData(stockData) =

let serializedData = stockData

|> Seq.map(fun row -> JsonConvert.SerializeObject(row))

File.WriteAllLines(filePath,serializedData)

member this.GetData() =

let serializedData = File.ReadAllLines(filePath)

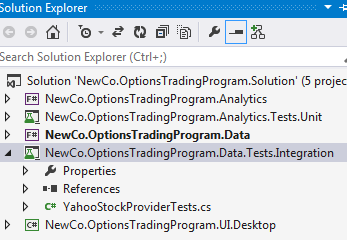
serializedData

|> Seq.map(fun row -> JsonConvert.DeserializeObject<(DateTime\*float\*float\*float\*float\*int\*float)>(row))

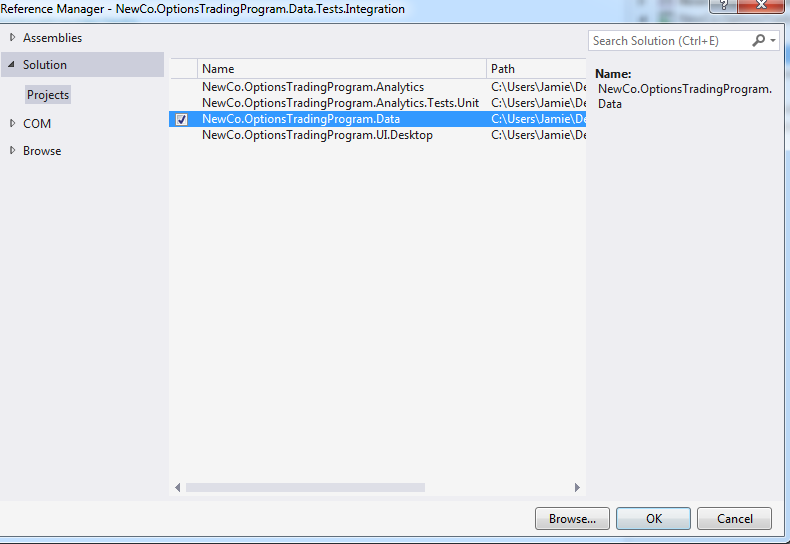
You will notice a subtle difference in the FileSystemStockProvider -> the filepath is now part of the constructor. This is so we can apply an interface to both the YahooStockProvider and the FileSystemStockProvider and swap them out at runtime, depending on our needs. Note that we are keeping the script file as part of the solution just to have. If we need to make additional changes to the code, we can always go into our script to “prove out” the concept and then bring the code into our module.

Next, let’s build a couple of integration tests so that other non-F# developers can see how to use our classes and so we have a repeatable way of verifying our code.

Open the newCo.OptionsTradingProgram.Data.Tests.Integration project and rename UnitTest1 to YahooStockProviderTests



Next, add a reference to the NewCo.OptionsTradingProgram.Data project from the test project:



Next, Update the YahooStockProviderTests class to test a call to yahoo.

[TestClass]

public class YahooStockProviderTests

{

[TestMethod]

public void YahooStockProviderUsingMSFT\_ReturnsExpected()

{

YahooStockProvider stockProvider = new YahooStockProvider();

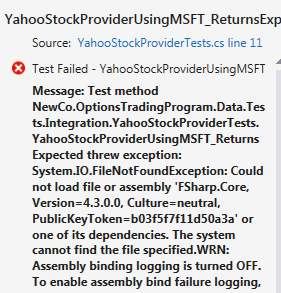
var data = stockProvider.GetData("MSFT");

Assert.IsNotNull(data);

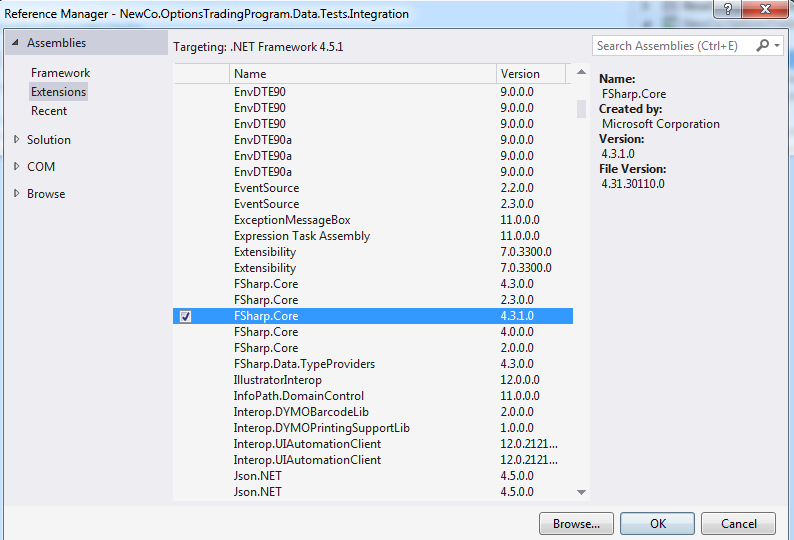
}

}

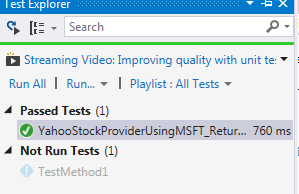
When you try and run it for the first time, it will fail with this message.



The problem is that FSharp has its own internal collection of types that are not part System.Core. The choice is to either add a reference to FSharp.core in the consuming application (like this C# test project) or to convert the FSharp types to a general .NET type. Since we are using type providers to allow inferred typing, we are going to do option #1. Open up the test project references (Assemblies->Extensions) and add a reference to FSharp.core (it might be 4.31 or 4.3.0 depending on your version of VS/Mono)



Now when you run the Test project, it now passes:



Add in another test class to the unit test project and call it FileSystemStockProviderTests. Add a couple of integration tests

[TestMethod]

public void PutData\_ReturnsExpected()

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

if (data != null)

{

provider.PutData(data);

}

}

[TestMethod]

public void GetData\_ReturnsExpected()

{

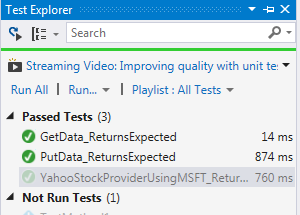
FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

Assert.IsNotNull(data);

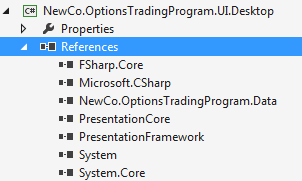
}

Run them to get green

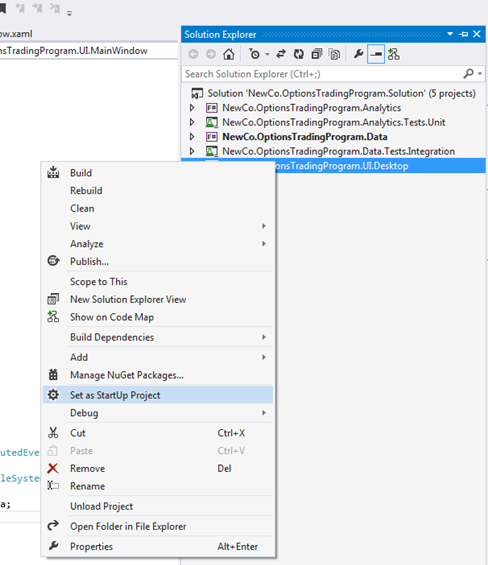


**Build a UI**

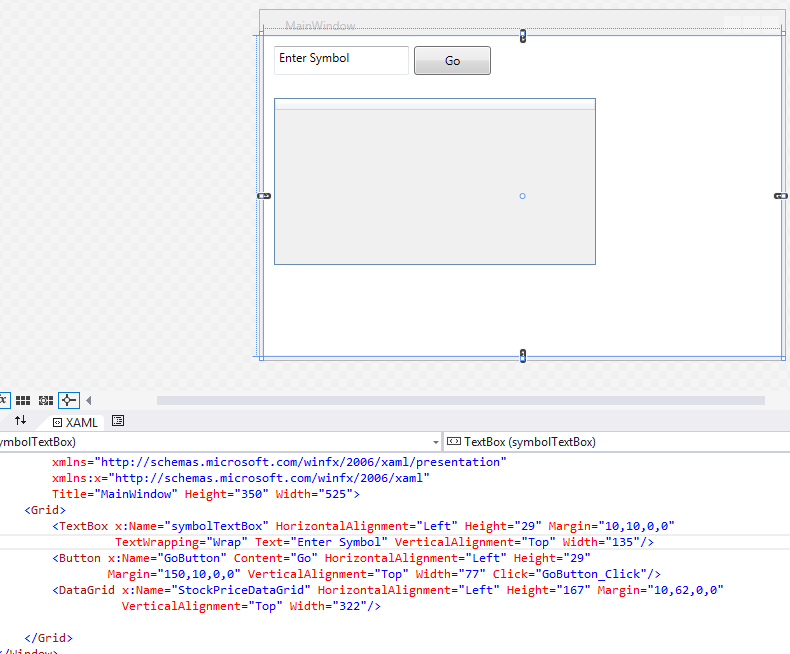
With the data layer now set, we are ready to build a UI that can consume our data/analytics. Open the NewCo.OptionsTradingProgram.UI.Desktop. Add a reference to NewCo.OptionsTradingProgram.Data and FSharp.Core



Next, set this project as the startup one:



In the Main Window designer, add a textbox to enter a symbol, a grid to display the data, and a button to get the data and put it in the grid

.

Go to the code behind and in the event handler for the button, add the following code:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

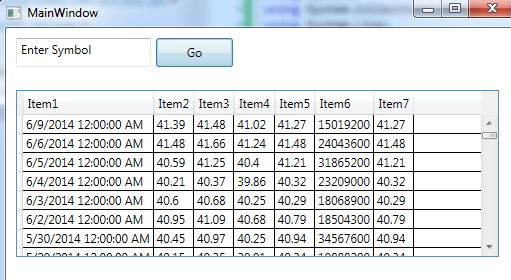
FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var data = provider.GetData();

this.StockPriceDataGrid.ItemsSource = data;

}

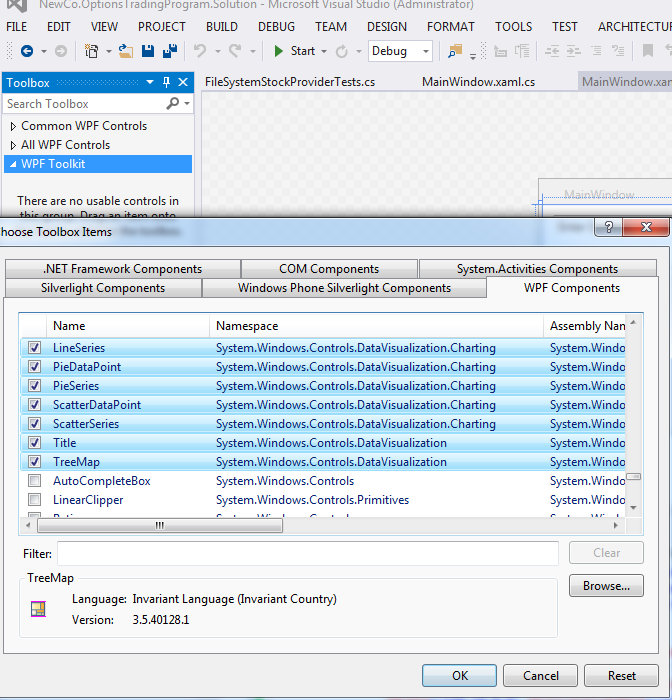
Run the project and you should get something like:



Note that you don’t need to enter the symbol b/c we are pulling static data from disk.

Next, install the WPF Toolkit from codeplex: <http://wpf.codeplex.com/releases/view/40535>

Back in the designer, open the toolbox and add a tab called WPF Toolkit. Right click and add all controls in the System.Windows.Controls.DataVisualization.Toolkit namespace. You can sort by assembly name in the dialog box



Back in the page designer, drag a bar series from the toolbox onto the canvas. Call it stockPriceLineGraph

<Grid Margin="0,0,0,-261">

<Grid.RowDefinitions>

<RowDefinition Height="77\*"/>

<RowDefinition Height="17\*"/>

</Grid.RowDefinitions>

<TextBox x:Name="symbolTextBox" HorizontalAlignment="Left" Height="29" Margin="10,10,0,0"

TextWrapping="Wrap" Text="Enter Symbol" VerticalAlignment="Top" Width="135"/>

<Button x:Name="GoButton" Content="Go" HorizontalAlignment="Left" Height="29"

Margin="150,10,0,0" VerticalAlignment="Top" Width="77" Click="GoButton\_Click"/>

<DataGrid x:Name="StockPriceDataGrid" HorizontalAlignment="Left" Height="167" Margin="10,62,0,0"

VerticalAlignment="Top" Width="483"/>

<chartingToolkit:Chart x:Name="chart" Width="350" Height="250">

<chartingToolkit:Chart.Series>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Black" VerticalAlignment="Top" Width="220"/>

</chartingToolkit:Chart.Series>

</chartingToolkit:Chart>

</Grid>

In the code behind, take the top 10 rows from the provider and add those points to the graph:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var stockPrices = provider.GetData().Take(10);

this.StockPriceDataGrid.ItemsSource = stockPrices;

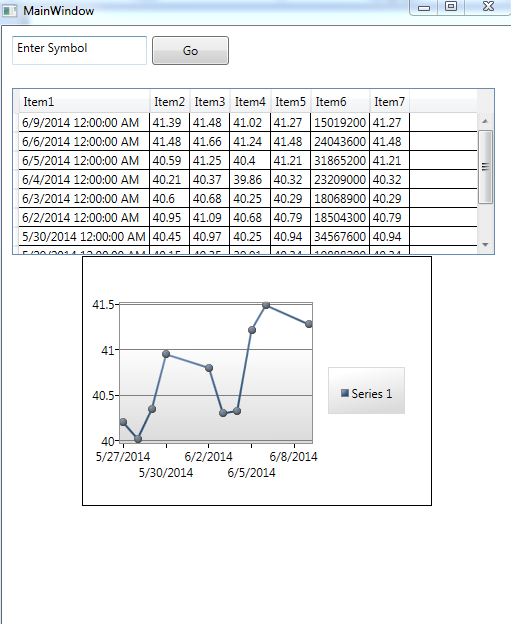
this.stockPriceLineGraph.DependentValuePath="Item7";

this.stockPriceLineGraph.IndependentValuePath = "Item1";

this.stockPriceLineGraph.ItemsSource = stockPrices;

}

And you should see:



**Implement Basic Statistics**

Open NewCo.OptionsTradingProgram.Analytics. Open the Script.fsx file and remove everything in it. There are some mathematical formulas we will need to use for our analysis. This is an area where F# really shines because it makes working with mathematical formulas very natural. In the script file, create an array of numbers 1 to 6 for analysis

let testData = [1.0 .. 6.0]

Hit ALT+ENTER and you can see the results in the REPL



In the script file, sum the numbers in the array

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData



Note that you can sum the numbers using a ‘pipe forward’ syntax. This is the syntax that I will be using by default from now on.

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum



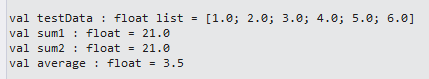
In the script file, average the numbers in the array

let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum

let average = testData |> Seq.average



In the script file, find the minimum value the numbers in the array

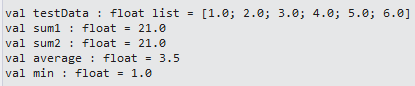
let testData = [1.0 .. 6.0]

let sum1 = Seq.sum testData

let sum2 = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min



In the script file, find the maximum value the numbers in the array

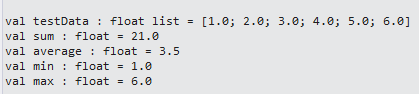
let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max



In the script file, find all even numbers in the array

let testData = [1.0 .. 6.0]

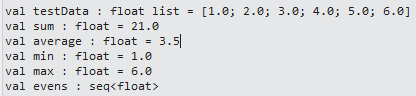
let sum = testData |> Seq.sum

let average = testData |> Seq.average

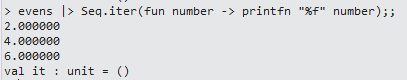
let min = testData |> Seq.min

let max = testData |> Seq.max

let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)



Note that the even numbers are in their own list. To actually see each member, go to the REPL and type this



In the script file, add 1 number to every number in the array

let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

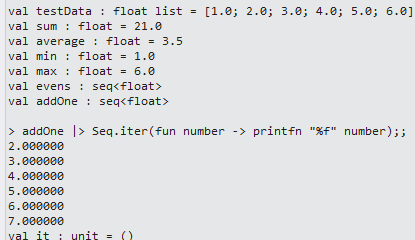
let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max

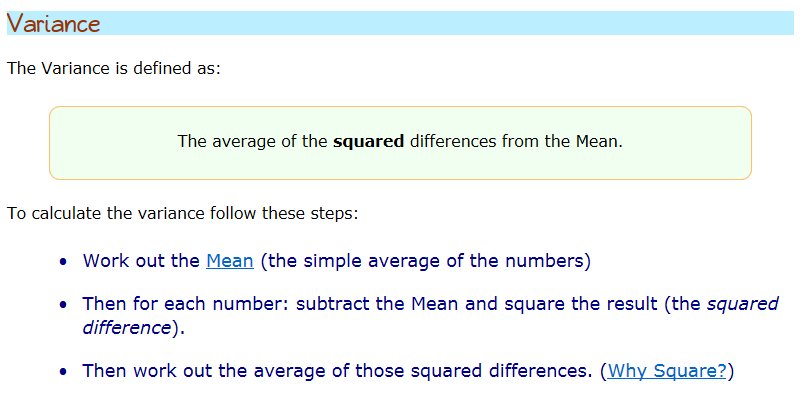
let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)

let addOne = testData |> Seq.map(fun number -> number + 1.)



Now let’s create the formula for variance. If you are not familiar with variance, here is a good description: http://www.mathsisfun.com/data/standard-deviation.html

And here is the methodology that the page uses



In the script file, create the variance. Notice how there is a one to one correspondence between F# and to the written steps:

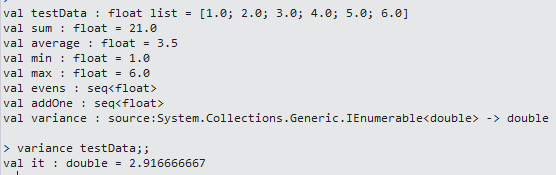
open System.Collections.Generic

let variance (source:IEnumerable<double>) =

let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas



Next, let’s calculate the standard deviation. You can refer back to that Math Is Fun page for a full explanation of standard deviation. The formula is: 

In our code, add in the standardDeviation:

open System.Collections.Generic

let testData = [1.0 .. 6.0]

let sum = testData |> Seq.sum

let average = testData |> Seq.average

let min = testData |> Seq.min

let max = testData |> Seq.max

let evens = testData |> Seq.filter(fun number -> number % 2. = 0.)

let addOne = testData |> Seq.map(fun number -> number + 1.)

//http://www.mathsisfun.com/data/standard-deviation.html

let variance (source:IEnumerable<double>) =

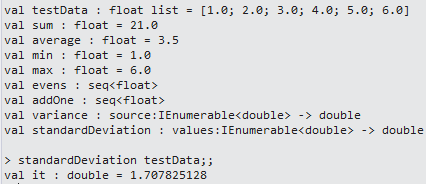
let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

let standardDeviation(values:IEnumerable<double>) =

sqrt(variance(values))



Next, let’s add in a moving average. If you are not familiar with moving average, here is a good explanation: <http://www.investopedia.com/terms/m/movingaverage.asp>

To calculate the moving average, follow these steps:

* Break a list of numbers into a series of smaller lists
* Calculate the average on those sublists

Again, there is a one to one correspondence between F# and how you would describe the calculation in English:

let movingAverage(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> Seq.average)



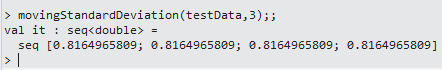
Next, let’s add a moving standard deviation. It is the same concept as a moving average and the code is identical except for 1 place:

let movingStandardDeviation(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> standardDeviation)



Next, let’s add a Bollinger bands. This is the first financial-related formula we have. If you are not familiar with Bollinger bands, a good explanation is here: <http://www.investopedia.com/terms/b/bollingerbands.asp>. To calculate Bollinger bands, you need to get 2 standard deviations away from a simple moving average:

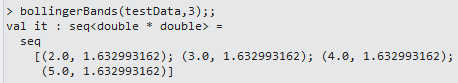
let bollingerBands (values:IEnumerable<double>, windowSize:int)=

let movingAverage = movingAverage(values,windowSize)

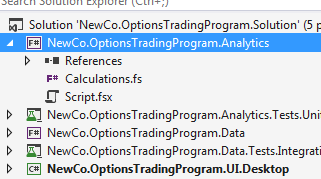
let movingStandardDeviation = movingStandardDeviation(values,windowSize)

let movingStandardDeviation' = movingStandardDeviation |> Seq.map(fun window -> window \* 2.)

Seq.zip movingAverage movingStandardDeviation'



Let’s take a break from our scripting and make a chart from our Bollinger bands. In the solution explorer, rename Library1.fs to Calculations.fs



Open Calculations.fs and rename Class1() to Calculations()

namespace NewCo.OptionsTradingProgram.Analytics

type Calculations() =

member this.X = "F#"

Next, add in the reference to System.Collections.Generic and the variance calculation

namespace NewCo.OptionsTradingProgram.Analytics

open System.Collections.Generic

type Calculations() =

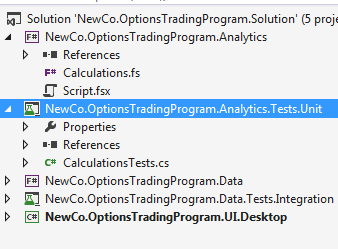
member this.Variance (source:IEnumerable<double>) =

let mean = Seq.average source

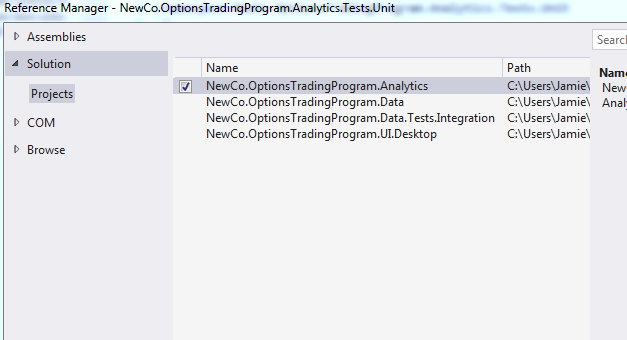
let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

Next, in the solution explorer, rename UnitTest1 to CalculationsTests



Open NewCo.OptionsTradingProgram.Analytics.Tests.Unit and add a reference to NewCo.OptionsTradingProgram.Analytics



Open CalculationsTests, rename TestMethod1 to CalculateVarianceUsingValidData\_ReturnsExpected and implement it like so

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using NewCo.OptionsTradingProgram.Analytics;

namespace NewCo.OptionsTradingProgram.Analytics.Tests.Unit

{

[TestClass]

public class CalculationsTests

{

[TestMethod]

public void CalculateVarianceUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var variance = calculations.Variance(testData);

var expected = 2.916666667;

var actual = Math.Round(variance, 9);

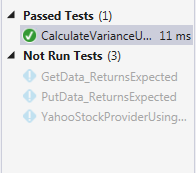
Assert.AreEqual(expected, actual);

}

}

}

Run your unit test and get green:



Go back to the Calculations.fs file and add in the remaining formulas from your script file. You will replace “let” with “member this.” and apply Pascal casing to the function names:

namespace NewCo.OptionsTradingProgram.Analytics

open System.Collections.Generic

type Calculations() =

member this.Variance (source:IEnumerable<double>) =

let mean = Seq.average source

let deltas = Seq.map(fun x -> pown(x-mean) 2) source

Seq.average deltas

member this.StandardDeviation(values:IEnumerable<double>) =

sqrt(this.Variance(values))

member this.MovingAverage(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> Seq.average)

member this.MovingStandardDeviation(values:IEnumerable<double>, windowSize:int)=

values

|> Seq.windowed (windowSize)

|> Seq.map(fun window -> window |> this.StandardDeviation)

member this.BollingerBands (values:IEnumerable<double>, windowSize:int)=

let movingAverage = this.MovingAverage(values,windowSize)

let movingStandardDeviation = this.MovingStandardDeviation(values,windowSize)

let movingStandardDeviation' = movingStandardDeviation |> Seq.map(fun window -> window \* 2.)

Seq.zip movingAverage movingStandardDeviation'

Compile, go to your test project, and add in the following tests:

[TestMethod]

public void CalcualteStandardDeviationUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var standadDeviation = calculations.StandardDeviation(testData);

var expected = 1.707825128;

var actual = Math.Round(standadDeviation,9);

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateMovingAverageUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var movingAverage = calculations.MovingAverage(testData,3);

var expected = 4;

var actual = new List<Double>(movingAverage).Count;

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateMovingStandardDeviationUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var movingStandardDeviation = calculations.MovingStandardDeviation(testData,3);

var expected = 4;

var actual = new List<Double>(movingStandardDeviation).Count;

Assert.AreEqual(expected, actual);

}

[TestMethod]

public void CalculateBollingerBandsUsingValidData\_ReturnsExpected()

{

var testData = new Double[6] { 1, 2, 3, 4, 5, 6 };

var calculations = new Calculations();

var bollingerBands = calculations.BollingerBands(testData,3);

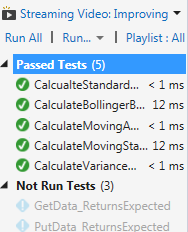
var expected = 4;

var actual = new List<Tuple<Double,Double>>(bollingerBands).Count;

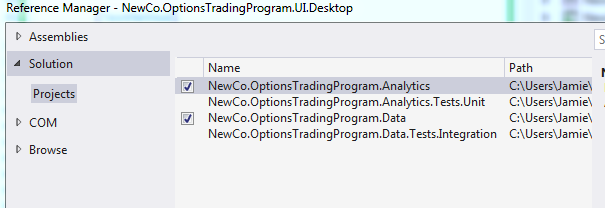
Assert.AreEqual(expected, actual);

}

Note you will have to add a reference to System.Collections.Generics. Once the tests are ready, run them and get green



Finally, open NewCo.OptionsTrandingProgram.UI.Desktop and add a reference to NewCo.OptionsTradingProgram.Analytics.



Open the MainWindow.xaml file and alter the chart to have three line series:

<chartingToolkit:Chart x:Name="chart" Width="350" Height="250">

<chartingToolkit:Chart.Series>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Black" VerticalAlignment="Top" Width="220"/>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph2"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Red" VerticalAlignment="Top" Width="220"/>

<chartingToolkit:LineSeries x:Name="stockPriceLineGraph3"

IsSelectionEnabled="True"

HorizontalAlignment="Left" Height="178"

Foreground="Green" VerticalAlignment="Top" Width="220"/>

</chartingToolkit:Chart.Series>

</chartingToolkit:Chart>

Open the MainWindow.xaml.cs file. Add a reference to NewCo.OptionsTradingProgram.Analytics. Next, alter the GoButton\_Click event handler like this:

private void GoButton\_Click(object sender, RoutedEventArgs e)

{

FileSystemStockProvider provider = new FileSystemStockProvider(@"C:\Data\TEST.txt");

var stockPrices = provider.GetData().Take(20);

this.StockPriceDataGrid.ItemsSource = stockPrices;

var adjustedClosePrices = from stockPrice in stockPrices

select stockPrice.Item7;

var dates = from stockPrice in stockPrices.Skip(2)

select new { stockPrice.Item1 };

var calculations = new Calculations();

var movingAverage = calculations.MovingAverage(adjustedClosePrices, 3);

var movingAverages = dates.Zip(movingAverage, (d, p) => new { date=d.Item1, price=p});

var bollingerBands = calculations.BollingerBands(adjustedClosePrices, 3);

var upperBandBands = dates.Zip(bollingerBands, (d, bb) => new { date = d.Item1, upperBand = bb.Item1 + (bb.Item2 \* 2) });

var lowerBandBands = dates.Zip(bollingerBands, (d, bb) => new { date = d.Item1, lowerBand = bb.Item1 + (bb.Item2 \* 2) \* -1 });

this.stockPriceLineGraph.DependentValuePath = "price";

this.stockPriceLineGraph.IndependentValuePath = "date";

this.stockPriceLineGraph.ItemsSource = movingAverages;

this.stockPriceLineGraph2.DependentValuePath = "upperBand";

this.stockPriceLineGraph2.IndependentValuePath = "date";

this.stockPriceLineGraph2.ItemsSource = upperBandBands;

this.stockPriceLineGraph3.DependentValuePath = "lowerBand";

this.stockPriceLineGraph3.IndependentValuePath = "date";

this.stockPriceLineGraph3.ItemsSource = lowerBandBands;

}

You will notice we are using Linq, which has almost identical syntax to F#. Indeed, if you understand F#, Linq because very accessible. Unfortunately, the reverse is not true. In any event, if you run the application, you will get something like this: