

Whitepaper

Python Analyses with gSharp and the AVEVA PI System



1 Table of Contents

2	Introduction.....	3
3	Requirements.....	4
4	Creating and building the project.....	5
4.1	Creating the gSharp project.....	5
4.2	Preparing the project.....	6
4.3	Writing the code.....	9
4.3.1	Variables.....	9
4.3.2	Module Initializer.....	9
4.3.3	Helper Class.....	10
4.3.4	Python script	11
4.3.5	Helper methods	13
4.3.6	Execute method.....	15
	Setting up the Testbed for debugging.....	16
4.4	Debugging.....	16
5	Deployment.....	18

2 Introduction

Using Python scripts for machine learning based analysis of data stored within the PI System is a common approach used by many PI System users. Python is often the language of choice for data scientists given its ease of access to libraries specifically designed for machine learning algorithms.

However, integration with the PI System as both a data source and destination for analysis results is the area where most users tend to resort to manual data extraction and preparation. This whitepaper details the use of gHost Technology's gSharp analytics platform as a means of integrating PI System data with existing Python scripts and providing a mechanism to utilise these as scheduled streaming analytics.

A practical step by step process of implementing a gSharp calculation to call a Python script and pass data between PI and Python is outlined in this whitepaper. All demonstration code in this whitepaper is provided 'as-is', and no warranty (express or implied) is given as to the fitness for purpose of any code. The code in this document has been tested and run as a demonstration, and you should add your own additional error handling as needed.

The sample project used in this whitepaper is available at <https://github.com/GTSGroupAustralia>. The SampleCode repo contains sample projects in both C# and VB.NET.

3 Requirements

The following software requirements are used in the examples provided in this whitepaper:

- Visual Studio – either 2019 or 2022 versions, and any of the Community, Professional, or Enterprise editions.
- gSharp developer components installed on the same machine as Visual Studio.
- gSharp server.
- Python version 3.7 or higher (up to Python 3.13), installed on both the gSharp server and the development machine where Visual Studio is installed.
 - Any Python libraries or packages explicitly required by your Python scripts.
- PI AF Client 2018 SP3 Patch 6 or higher.

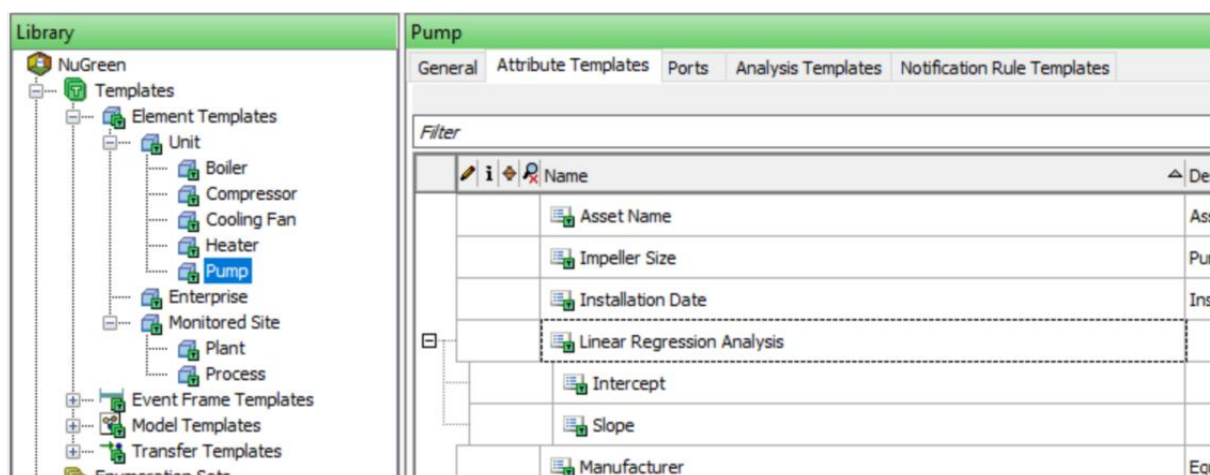
The example code in this whitepaper is written in C# targeting language version 12.0. By default, .NET Framework 4.8 uses C# language version 7.3. To change the language version in your project, open the .csproj file in a text editor and add a <LangVersion> element in the first PropertyGroup node:

```
<TargetFrameworkVersion>v4.8</TargetFrameworkVersion>
<LangVersion>12.0</LangVersion>
```

It is assumed that a functioning gSharp installation is in place, as this whitepaper does not cover the installation and configuration of the gSharp components. For information on this, please refer to the documentation at <https://docs.ghost.site/gSharp/Getting%20Started/Installation/Introduction>.

The example calculation will be built against the demo NuGreen AF database. This AF database can be downloaded from AVEVA using the following link: <https://docs-be.aveva.com/bundle/pi-web-api-reference-1.19.1/page/static/help/NuGreen.xml>. Of course, you can also substitute your own existing AF database to test this code against.

For the purpose of this example, the following new attributes have been added to the Pump template in the NuGreen database:

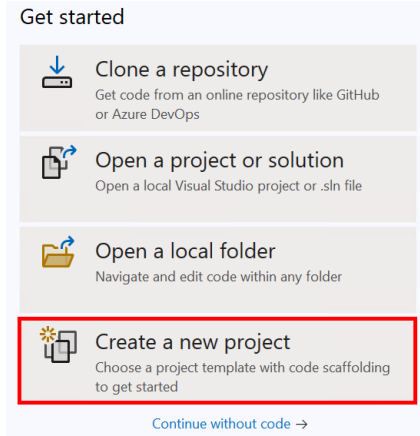


The Intercept and Slope attributes under Linear Regression Analysis will be outputs for our calculation and are defined as Single data type. The input will be the Process Feedrate attribute in the parent Unit element template.

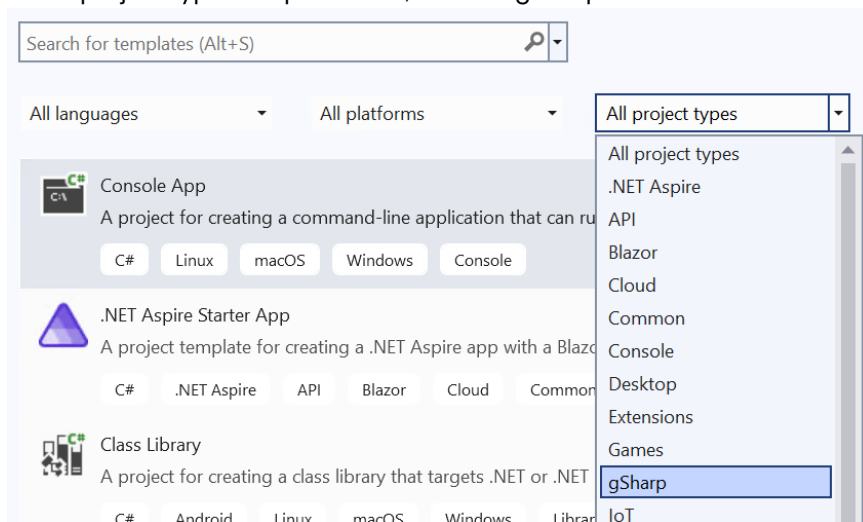
4 Creating and building the project

4.1 Creating the gSharp project

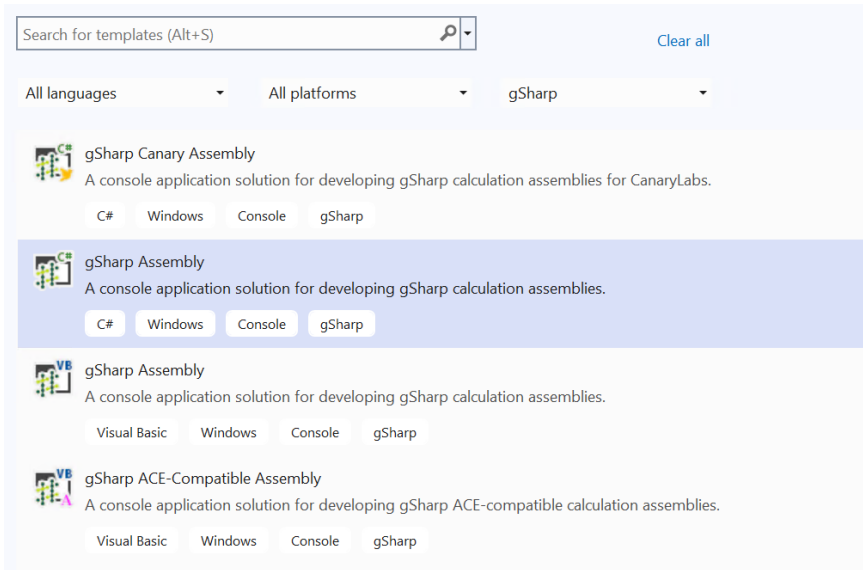
Open Visual Studio and select the Create a new project option:



In the project types dropdown list, filter for gSharp:



From the filtered results list, select the gSharp Assembly project for your choice of language (C# or Visual Basic), then click the Next button. The example project in this whitepaper uses C#.



Search for templates (Alt+S) Clear all

All languages All platforms gSharp

gSharp Assembly
A console application solution for developing gSharp calculation assemblies.
C# Windows Console gSharp

gSharp Assembly
A console application solution for developing gSharp calculation assemblies.
Visual Basic Windows Console gSharp

gSharp ACE-Compatible Assembly
A console application solution for developing gSharp ACE-compatible calculation assemblies.
Visual Basic Windows Console gSharp

Give your project a meaningful name, select the location where you wish to store the project, and ensure that the checkbox is selected for the 'Place solution and project in the same directory' option, then click the Create button:

Configure your new project

gSharp Assembly

C#

Windows

Console

gSharp

Project name

GSharp.PythonRegressionAnalysis

Location

C:\Users\john.messinger\source\repos

Solution name ⓘ

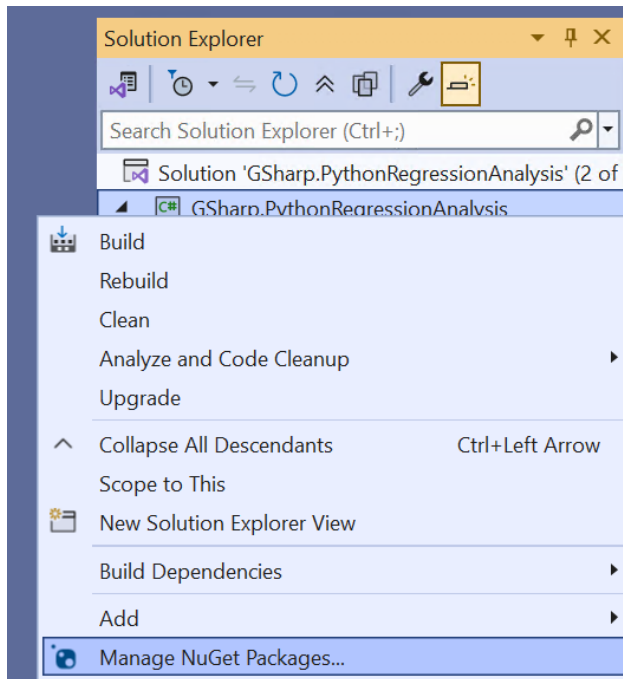
GSharp.PythonRegressionAnalysis

☒ Place solution and project in the same directory

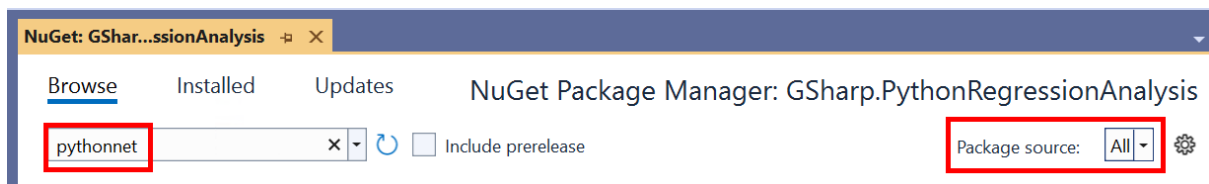
4.2 Preparing the project

Before writing any code, we first need to add a NuGet package to handle the Python interactions from the .NET environment. For this, we will use the open source Python.NET library (aka pythonnet) – see <https://pythonnet.github.io/> for more information.

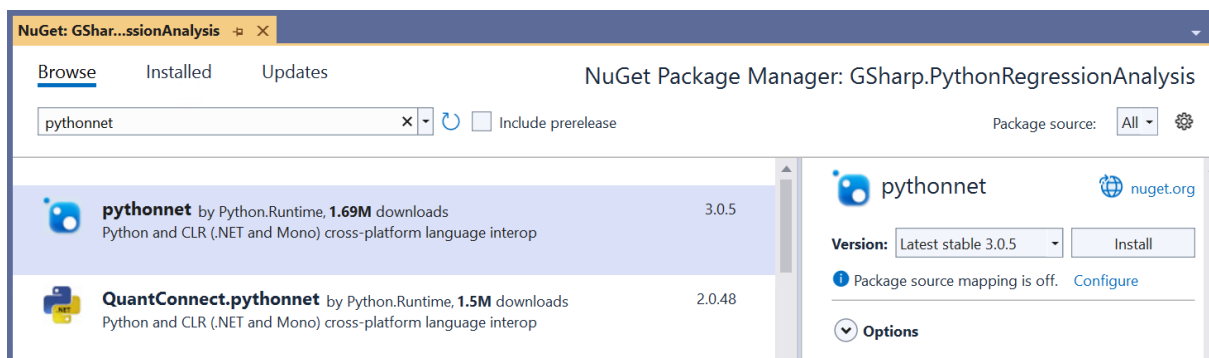
In the Solution Explorer pane in Visual Studio, right click on the calculation project and click the Manage NuGet Packages option in the context menu:



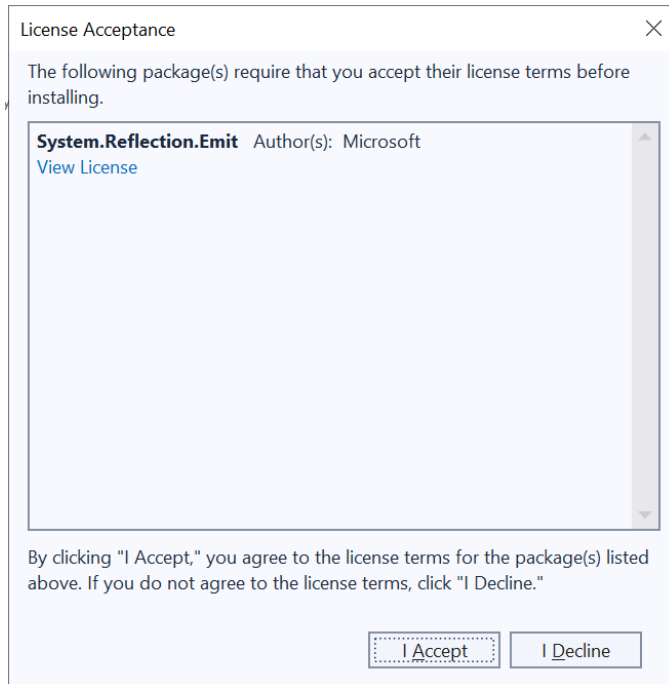
In the Browse tab of the NuGet Package Manager window, type 'pythonnet' (without the single quotes) in the search box, and change the package source from nuget.org to All:



From the results list, select pythonnet (by Python.Runtime) and install it:



This will also require the System.Reflection.Emit package from Microsoft as a dependency, so install that as well when prompted:

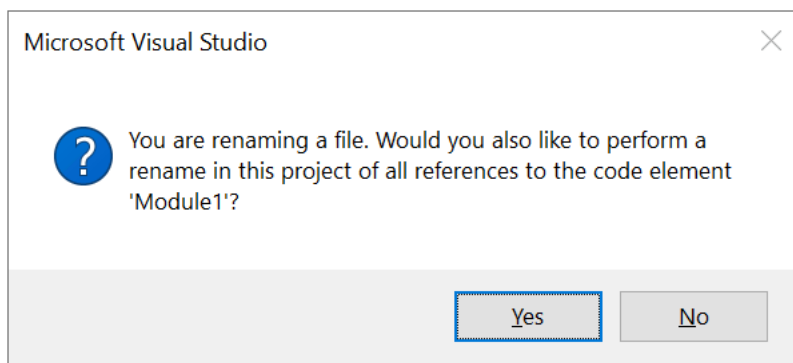


Once the NuGet packages are installed, close the NuGet package Manager window and return to the project.

You will also need to add a reference in your project to the AF SDK as the example code will retrieve data from your AF element context as an AFValues collection.

Lastly, create a new folder called PythonScripts in the project using the Solution Explorer. This is where we will store the Python script later on.

Using the Solution Explorer pane, right click on Module1.cs in your project and rename it to something meaningful, such as LinearRegression. You should be prompted with a dialog like the following:



Click Yes to ensure that all references to Module1 are properly renamed throughout the solution.

4.3 Writing the code

4.3.1 Variables

For this example, we need one input point, and two output points to store the results of the calculation.

- Change the name of the default MyPoint1 to ProcessFeedrate.
- Change the name of the default MyPoint2 to Intercept.
- Add a new variable of type AFDatapoint called Slope and set the PointDataDirection to DataDirection.Output.

```
public class LinearRegression : CalculationModuleAFBase
{
    [PointDataDirection(DataDirection.Input)]
    3 references
    public AFDatapoint ProcessFeedrate { get; set; }

    [PointDataDirection(DataDirection.Output)]
    2 references
    public AFDatapoint Intercept { get; set; }

    [PointDataDirection(DataDirection.Output)]
    0 references
    public AFDatapoint Slope { get; set; }
}
```

4.3.2 Module_INITIALIZER

Because of the single threaded nature of the Python interpreter and its use of a global interpreter lock (GIL), we need to set a property in the assembly that will instruct the calculation queue manager how to deal with this. In the assembly's `Initialize()` method, set the `IsSequential` property to true. This will instruct the queue manager to run multiple calculation contexts for this assembly sequentially rather than concurrently, thus avoiding the issues that will invariably occur with the Python interpreter.

We also need to specify the location of the installed Python interpreter so that the Python.NET bridge knows where to find it:

```
0 references
public override void Initialize()
{
    IsSequential = true;
    string pythonDll = @"C:\Program Files\Python\Python312\python312.dll";
    Environment.SetEnvironmentVariable("PYTHONNET_PYDLL", pythonDll);
}
```

During debugging and testing the path to the Python dll will be local to your development machine. If Python is installed to a different location on the gSharp server make sure that the path in the `Initialize` method is relative to the server before you compile and deploy the assembly.

4.3.3 Helper Class

Before adding code to the Execute method, we need to create a helper class used for formatting of the PI data into a JSON dictionary format to be passed to Python.

```
internal class JsonDataDictionary
{
    2 references
    public List<string> Date { get; private set; }

    2 references
    public List<string> Value { get; private set; }

    1 reference
    public JsonDataDictionary(List<AFValue> data)
    {
        Date = [];
        Value = [];

        foreach (var val in data)
        {
            Date.Add(val.Timestamp.ToString("yyyy-MM-ddTHH:mm:ss"));
            Value.Add(val.ValueAsString().ToString());
        }
    }
}
```

This class will be serialised using the Serialize method of the System.Text.Json.JsonSerializer class. When called, this will produce a JSON dictionary of data that looks like the following:

```
{
  "Date": [
    "2025-08-17",
    "2025-08-18",
    "2025-08-19",
    "2025-08-20",
    "2025-08-21"
  ],
  "Value": [
    94.6085433959961,
    87.4042129516602,
    77.4614715576172,
    65.7402114868164,
    53.0103607177734
  ]
}
```

4.3.4 Python script

In this example, we will use a simple Python script that calculates a linear regression over a set of timeseries data extracted from PI. The Python script is written as follows:

```
import json
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

def linear_regression_time_series(jdict):
    # Create a DataFrame from the dictionary
    df = pd.DataFrame(jdict)

    # Convert dates to ordinal values for regression
    df['Date_ordinal'] = pd.to_datetime(df['Date']).map(pd.Timestamp.toordinal)

    # Prepare the data for linear regression
    X = df['Date_ordinal'].values.reshape(-1, 1)
    y = df['Value'].values

    # Create and fit the model
    model = LinearRegression()
    model.fit(X, y)

    # Make predictions
    df['Predicted'] = model.predict(X)

    # Return results
    return model.intercept_, model.coef_[0]

def main(data):
    jdict = json.loads(data)
    calc_results = linear_regression_time_series(jdict)
    return calc_results

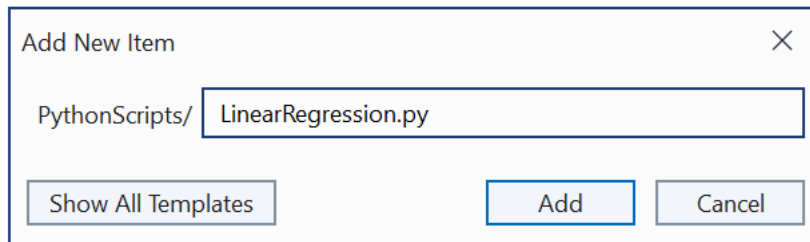
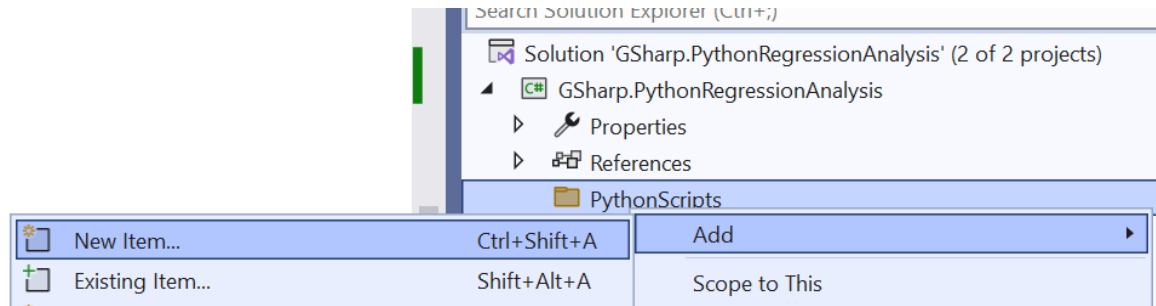
# Call the function with the sample data
result = main(data)
intercept = result[0]
slope = result[1]
```

Note that there is a main function defined that receives one parameter and returns an array result. The *data* variable passed to the main function is used to receive PI data from gSharp, and the *result* array is used to read the results from the Python environment into gSharp, to be written back to PI.

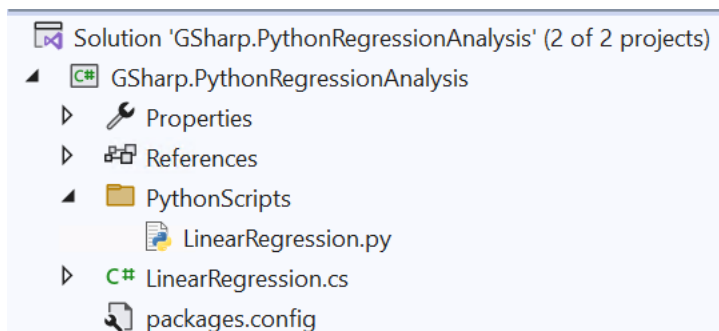
When using your own Python scripts, it's important to note that these need to be structured in a similar manner to the example script above:

- Defining a main function as the entry point into the script, with a parameter variable to receive the data from gSharp.
- Defining additional functions to manage the methods that the script will use to produce the required results.
- Defining one or more variables to hold the output results of the script that can be read by gSharp.

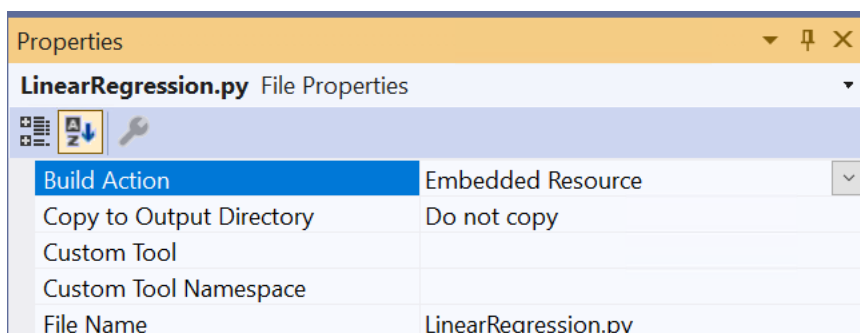
Using the Solution Explorer panel, right click on the PythonScripts folder previously created in the project and click Add > New Item



After creating the new Python script file, add the Python script from the previous page to this file and save it.



In this example we will set the Build Action of the Python script to 'Embedded Resource' so that everything is wrapped up in the assembly:



You will need to ensure that you have both the pandas and Scikit-learn packages installed for Python. Use the Pip package manager to install these.

(see <https://python.land/virtual-environments/installing-packages-with-pip>)

4.3.5 Helper methods

There are two helper methods that need to be created that will be called from the main Execute method.

4.3.5.1 ReadPythonScript method

To read the embedded Python script into a variable that can be passed to Python, we need to create a helper method as follows:

```
private string ReadPythonScript(string fileName)
{
    var assembly = System.Reflection.Assembly.GetExecutingAssembly();
    string resourcePath = fileName;
    resourcePath = assembly.GetManifestResourceNames()
        .Single(str => str.EndsWith(fileName));

    using Stream stream = assembly.GetManifestResourceStream(resourcePath);
    using StreamReader reader = new(stream);
    return reader.ReadToEnd();
}
```

4.3.5.2 RunPythonCodeAndReturn method

This method is where the Python interpreter will be invoked to execute our script with the data retrieved from PI.

```
private double[] RunPythonCodeAndReturn(string pycode, object dataset)
{
    try
    {
        PythonEngine.Initialize();

        double[] returnedVariable = new double[2];

        using (Py.GIL())
        {
            using (var scope = Py.CreateScope())
            {
                PyObject pyInputData = dataset.ToPython();
                scope.Set("data", pyInputData);
                scope.Exec(pycode);

                scope.TryGet<object>("slope", out object slope);
                scope.TryGet<object>("intercept", out object intercept);

                returnedVariable[0] = Convert.ToDouble(slope);
                returnedVariable[1] = Convert.ToDouble(intercept);
            }
        }

        PythonEngine.Shutdown();
        return returnedVariable;
    }
    catch (Exception ex)
    {
        Log.Error(ex);
        Py.GIL().Dispose();
        PythonEngine.Shutdown();
        throw;
    }
}
```

This method returns a double array as two results variables from Python are received – the calculated slope and y-intercept of the linear regression. Two parameters are passed to the method – a string variable holding the Python script, and an object representing the PI data in the form of a JSON dictionary.

Calling `PythonEngine.Initialize()` will initialize Python for use by our code. Before interacting with any of the objects or APIs provided by the `Python.Runtime` namespace, calling code must have acquired the Python global interpreter lock by using `Py.GIL()`. The only exception to this rule is the `PythonEngine.Initialize()` method, which may be called at startup without having acquired the GIL. The GIL is released again by disposing the return value of `Py.GIL()`. In the case of our gSharp example, we can wrap this into a standard C# using statement which will automatically call the `Dispose` method of the GIL upon completion of the code block.

Within the GIL using code block we need to create an instance of a `PyModule` object, which provides a Python scope for our script to execute within. Again, this is created within a using block so that it can be automatically disposed when our code has completed execution.

The JSON dictionary dataset passed into this method then needs to be converted to a Python object (`PyObject`). We can then pass this data directly into the Python scope using the `Set` method. This method takes two parameters, the first being the name of the script variable that will receive data, and the second parameter being the `PyObject` containing the actual data.

```
PyObject pyInputData = dataset.ToPython();
scope.Set("data", pyInputData);
```

The string "data" is the name of the variable in the Python script that is passed to the script's main function:

```
def main(data):
    jdict = json.loads(data)
    calc_results = linear_regression_time_series(jdict)
    return calc_results

# Call the function with the sample data
result = main(data)
```

The scope's `Exec` method is then called, with a single parameter passed in, being the string variable containing the actual Python script to be executed by the interpreter.

Once the script has been successfully executed, we can read out of the Python scope the return variables using the `TryGet` method. This method signature is `TryGet<T>` (string name, out T? value)

```
scope.TryGet<object>("slope", out object slope);
scope.TryGet<object>("intercept", out object intercept);

returnedVariable[0] = Convert.ToDouble(slope);
returnedVariable[1] = Convert.ToDouble(intercept);
```

The objects returned from the Python scope are converted to doubles to be returned to the `Execute` method.

Finally, after the GIL has been disposed, we need to explicitly call the `Shutdown()` method of the `PythonEngine` to release all resources being held by the Python runtime.

4.3.6 Execute method

The Execute method is the main entry point to the calculation, and here we will retrieve the input data from PI, format it into a JSON dictionary to be passed into Python, pass both the data and the Python script to the Python engine, and finally write our calculation results back to PI.

We will use the SampledValues method of the ProcessFeedrate AFDatapoint variable to retrieve 5-minute sampled data for the previous eight hours. You can also use the RecordedValues method to retrieve raw archive data.

```
AFValues tsDataset = ProcessFeedrate.SampledValues("*-8h", "*", "5m");

// Clean the data before further processing, i.e., remove any Bad values
var cleanDataSet = tsDataset.Where(v => v.IsGood).ToList();

var pyDictionary = new JsonDataDictionary(cleanDataSet);
string pyDataset = JsonSerializer.Serialize(pyDictionary);
```

We then read the embedded Python script into a string variable and pass that plus the JSON dictionary of PI data to the RunPythonCodeAndReturn method.

Once this method returns the results from Python, they can be written back to PI by assigning the value to the Value property of the Intercept and Slope AFDatapoint output points.

The complete code for the Execute method is as follows:

```
public override void Execute(ISchedule schedule, DateTime timestamp, CancellationToken cancelToken)
{
    try
    {
        AFValues tsDataset = ProcessFeedrate.SampledValues("*-8h", "*", "5m");

        // Clean the data before further processing, i.e., remove any Bad values
        var cleanDataSet = tsDataset.Where(v => v.IsGood).ToList();

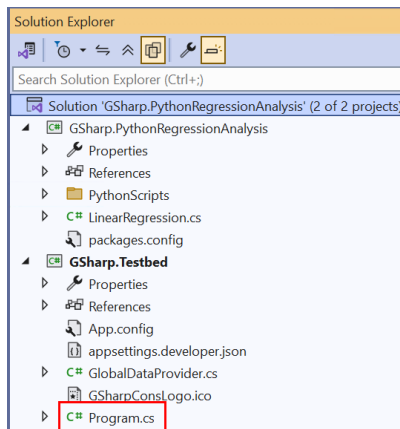
        var pyDictionary = new JsonDataDictionary(cleanDataSet);
        string pyDataset = JsonSerializer.Serialize(pyDictionary);
        string code = ReadPythonScript("LinearRegression.py");

        double[] result = RunPythonCodeAndReturn(code, pyDataset);

        Intercept.Value = result[0];
        Slope.Value = result[1];
    }
    catch (Exception ex)
    {
        Log.Error(ex);
    }
}
```

Setting up the Testbed for debugging

Before this assembly can be debugged, an AF Element context needs to be provided for the testbed application. Open the Program.cs file in the GSharp.testbed project.



In this file, the ConfigureAssembly method needs to be updated to map the AFDataPoint variables to AF Attribute names, and an AF Element context needs to be defined:

```
private void ConfigureAssembly(AssemblyBuilder builder)
{
    builder.AddModule<LinearRegression>(moduleBuilder => moduleBuilder
        .ConfigurePoint(nameof(LinearRegression.ProcessFeedrate), "|Attribute1")
        .ConfigurePoint(nameof(LinearRegression.Intercept), "|Attribute2")
        .ConfigurePoint(nameof(LinearRegression.Slope), "|Attribute2")
        .AddContext("ContextName1", @"\\AFServer\AFDatabase\Parent\Element 001")
    );
}
```

For the example code using the NuGreen database, the updated code in this method should look as follows:

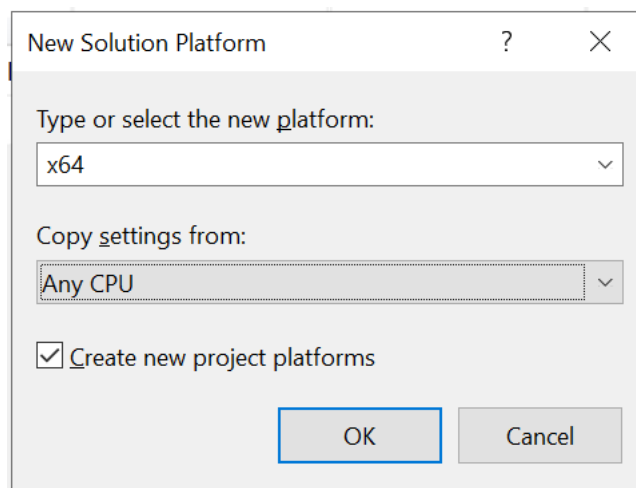
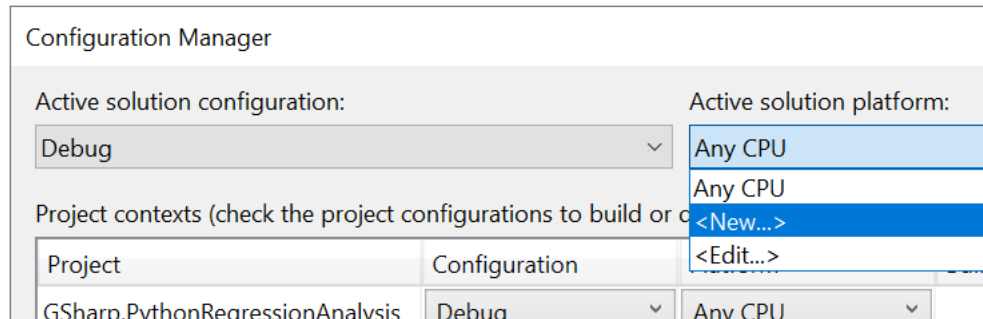
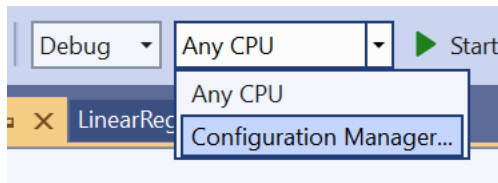
```
private void ConfigureAssembly(AssemblyBuilder builder)
{
    builder.AddModule<LinearRegression>(moduleBuilder => moduleBuilder
        .ConfigurePoint(nameof(LinearRegression.ProcessFeedrate), "|Process Feedrate")
        .ConfigurePoint(nameof(LinearRegression.Intercept), "|Linear Regression Analysis|Intercept")
        .ConfigurePoint(nameof(LinearRegression.Slope), "|Linear Regression Analysis|Intercept")
        .AddContext("ContextName1", @"\\SALES-AF1\NuGreen\NuGreen\Tucson\Distilling Process\Equipment\P-871")
    );
}
```

The context name is unimportant, but the path to the AF Element must be a valid fully qualified element path that includes both AF server name and AF database name. Once the attribute and element path mappings are complete, the code is ready for debugging.

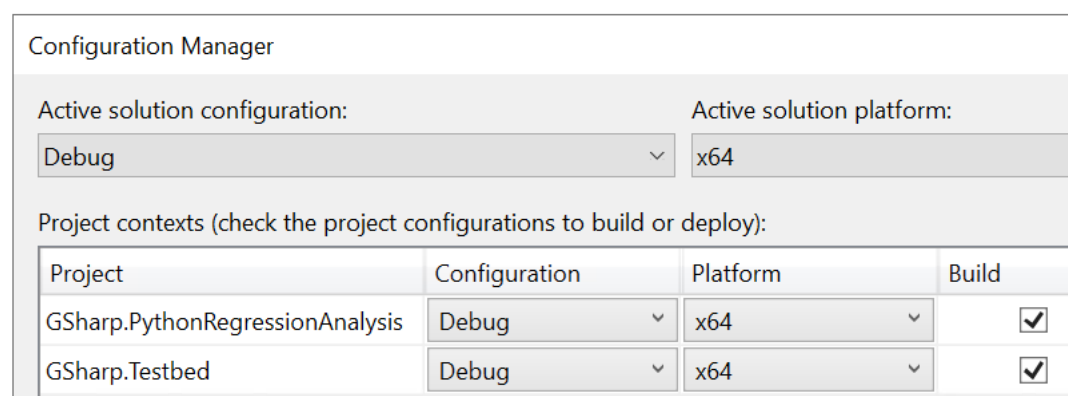
4.4 Debugging

Before you start debugging this assembly, one last change needs to be made to the project. Using the Python.NET bridge requires that the target processor architecture be updated to x64. The 'Any CPU' target will not work with Python.NET and the Python interpreter.

From the toolbar, open the Configuration Manager to create a new configuration:



You should end up with the following:



The project is now ready for debugging as required. Note that if you are stepping through the code in the `RunPythonCodeAndReturn` method, when control is passed to the Python interpreter after calling `scope.Exec(pycode)` that you can't step through the actual Python script as control has been passed to an external process.

5 Deployment

Once satisfied with the code, compile as a Release build (using the x64 target) and deploy to the gSharp server. The assembly file (GSharp.PythonRegressionAnalysis.dll) and the NuGet dependency (Python.Runtime.dll) will both need to be copied to the gSharp staging folder for deployment.

For detailed instructions on assembly deployment, please refer to the online documentation at <https://docs.ghost.site/gSharp/Getting%20Started/CalculationDevelopment/AssemblyRegistrationAndConfiguration>.