The Python plugin plugin
PRINTED MANUAL

Python plugin plugin

Printed: 11/2/2024

Publisher

AGG Software

Production

©2018-2024 AGG Software
http://www.aggsoft.com

Table of Contents

Part 1 Introduction 1
Part 2 System requirements 1
Part 3 Installing Python plugin 1
Part 4 Glossary 2
Part 5 Setup 3
Part 6 API 3
Part 7 Buffer 5
Part 8 List of values 5
Part 9 Built-in functions 7
Part 10 Module parameters 8

1 Introduction

Based on this module, you can create your own extension using the Python 3 programming language. Python plugin provides interaction of the main program and Python via a simple API.

Scripts can function as modules of data query, data parser, data filter, data export, and events handler. One script can perform several tasks.

Scripts have access to all features of Python. These can be either simple scripts to change a parser variable value, or more complex scripts with timers, threads, and system calls.

After installation, in the module's folder, you can find a few examples of use for different application areas.

2 System requirements

The following requirements must be met for "Python plugin" to be installed:

Operating system: Windows 2000 SP4 and above, including both x86 and x64 workstations and servers. The latest service pack for the corresponding OS is required.

Free disk space: Not less than 5 MB of free disk space is recommended.

Special access requirements: You should log on as a user with Administrator rights in order to install this module.

The main application (core) must be installed, for example, Advanced Serial Data Logger.

3 Installing Python plugin

1. Close the main application (for example, Advanced Serial Data Logger) if it is running;
2. Copy the program to your hard drive;
3. Run the module installation file with a double click on the file name in Windows Explorer;
4. Follow the instructions of the installation software. Usually, it is enough just to click the "Next" button several times;
5. Start the main application. The name of the module will appear on the "Modules" tab of the "Settings" window if it is successfully installed.

If the module is compatible with the program, its name and version will be displayed in the module list. You can see examples of installed modules on fig.1-2. Some types of modules require additional configuration. To do it, just select a module from the list and click the "Setup" button next to the list. The configuration of the module is described below.

You can see some types of modules on the "Log file" tab. To configure such a module, you should select it from the "File type" list and click the "Advanced" button.

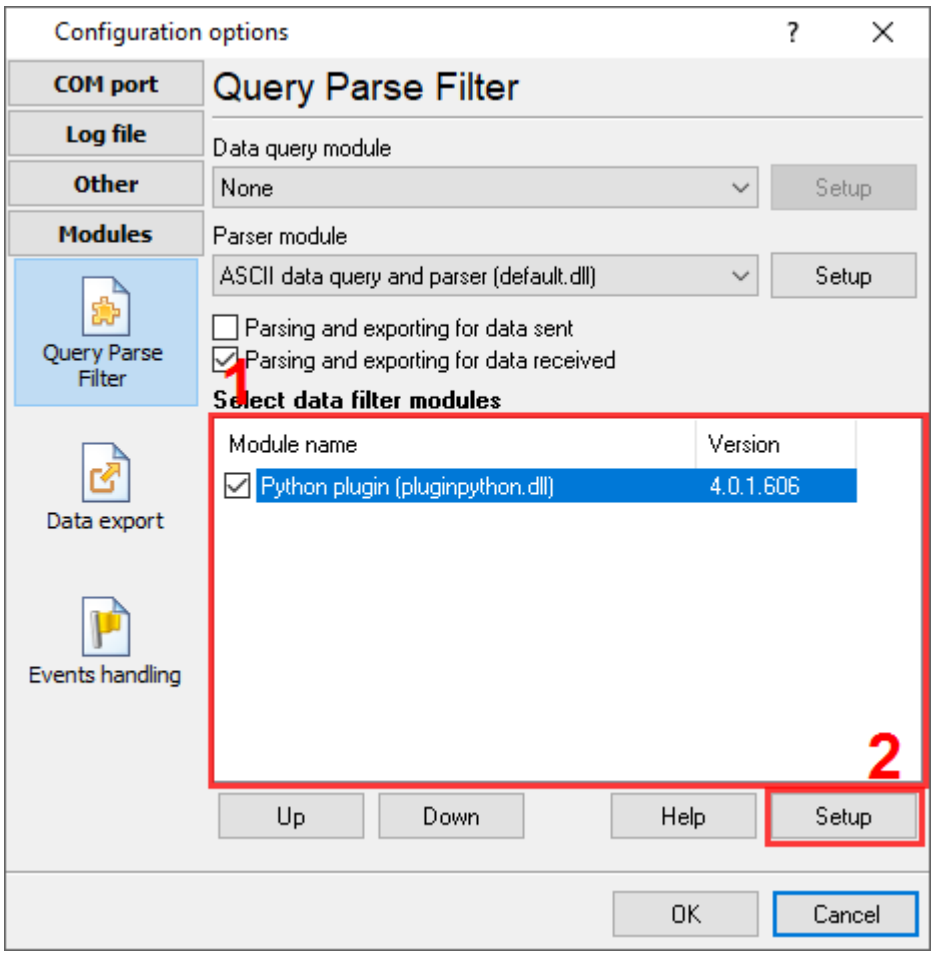

Fig. 1. Example of installed module

4 Glossary

Main program - it is the main executable of the application, for example, Advanced Serial Data Logger and asdlog.exe. It allows you to create several configurations with different settings and use different plugins.

Plugin - it is the additional plugin module for the main program. The plugin module extends the functionality of the main program.

Parser - it is the plugin module that processes the data flow, singling out data packets from it, and then variables from data packets. These variables are used in data export modules after that.

Core - see "Main program."

5 Setup

1. Download and install Python 3 for 32 bit systems (x86) on your computer.
2. You need to create a script in your editor because the module does not have its own script editing tools.
3. Specify the full name of the script file (Fig. 1).
4. Specify the path to python3x.dll (if the program could not detect the path automatically).

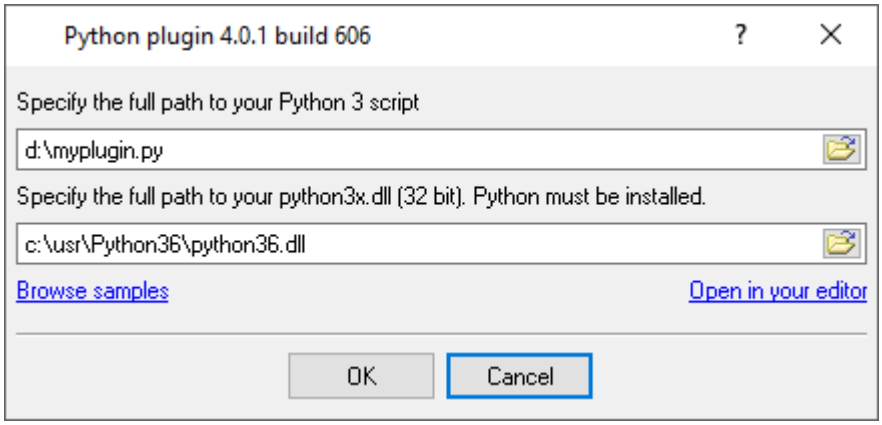

Fig. 1. Settings window

6 API

After installation, in the module's folder, you can find a few examples of use for different application areas. You can use one of the examples as a prototype.

Make sure your script code contains one or more functions with the specified name. When writing your script, you should keep in mind that the code is supposed to run as fast as possible. Using infinite loops is not allowed.

```
def FilterData(callparams, datasource, datain, dataout):
```

The main program calls this function if the "Python plugin" module is selected in the list of data filtering modules. The main program calls this function each time it receives a set of variables from the parser.

callparams - list of arguments to call the module.
datasource - data source identifier (it can be None).
datain - list of input data. It is a one-dimensional list (array) of parser variables, which can be divided into virtual strings with a special separator. See the examples of how to work with such an array.
dataout - list of output data.

This function can both modify an input list and create a completely new one. If values have been added to **dataout**, changes to **datain** will be ignored.

```
def ParseData(callparams, datasource, buffer, dataout):
```

The main program calls this function if the "Python plugin" module is selected in the list of data processing (parsing) modules. The main program calls this function each time it receives data from a data source and puts it to buffer. Please note that the program can receive data in small chunks, and calling this function does not always mean that a data packet has been received completely. If a data packet was not fully received, another function call will follow later.

callparams - list of arguments to call the module.
datasource - data source identifier (it can be None). By this identifier, a parser can distinguish data coming from different clients.
buffer - access object that allows accessing input data buffer. The program allocates small buffers (65 KB) for each connected data source. This buffer automatically accumulates all input data. The task of the parser is to remove processed data from the buffer. Otherwise, it will get overfilled, causing old data to be superseded by new ones.
dataout - list of output data.

In the case of successful processing of a data packet, the function must put at least one value in the **dataout** array.

The main program expects each parser to return three required variables:

FULL_DATA_PACKET - full data packet
DATA_PACKET - short data packet (may be the same as FULL_DATA_PACKET). Usually contains a data portion of the packet, without signatures, checksums, etc.
DATA_TIME_STAMP - returns date and time of data processing (usually datetime.datetime.now ()).

```
def ExportData(callparams, datasource, datain):
```

The main program calls this function if the "Python plugin" module is selected in the list of data export modules. The main program calls this function after calling the parser and data filtering modules. This is the last module in the chain of information handling. Therefore, the main program does not expect output data.

callparams - list of arguments to call the module.
datasource - data source identifier (it can be None).
datain - list of input data. Similar to FilterData.

```
def EventData(event_params, datasource, event):
```

This function is called if the "Python plugin" module has received an event that could be generated by the main program or another module. When calling the function, the main program does not expect any output data.

event_params - list of arguments to call the module. The list passes variables that make up the context of this event.
datasource - data source identifier (it can be None).
event - string value of event identifier.

```
def main():
```

The code of this function can be executed when loading and initializing the script. The code is run only once. This function lets you initialize any global variables. At the end of your script, you need to add a string of explicit invoking this function.

Example:

```
def main():

    print('Loaded')

main()
```

7 Buffer

This object has the following buffer interface functions.

Clear() - clears the buffer completely.

Shift(number) - deletes"**number**" bytes at the beginning of buffer (shifts the buffer).

GetBytes() - returns bytes from buffer as a standard list with "bytes" data type. Please note that if the function **Shift** was called after calling **GetBytes**, then the object "bytes" will keep the "obsolete" copy of the buffer.

Example:

```
>>> buffer.Clear()

>>> buffer.Shift(100)

>>> data = buffer.GetBytes()
```

8 List of values

A list of values is a set of items. Each item in the list is an object that has a name and a value. This list can be divided into one or more virtual strings by a special delimiter element, that has a fixed name "NEW_ROW."

The list has the following methods and properties.

Clear() - clears the list completely.

>>> list.Clear()

Delete(index) - removes an element with **index** from the list.

>>> list.Delete(0)

CopyFrom(object [, start_index = 0[, end_index = -1]]) - copies the specified number of items from the object list. Additional **start_index** and **end_index** parameters specify the start and end indexes of items in the source list. If **end_index** is -1, then all items in the list are copied down to the end.

```
>>> list1.CopyFrom(list2, 0, 10)
```

ItemByName(name) - returns a list item by its **name** (string).

```
>>> item = list.ItemByName('VALUE')
```

ItemIndexByName(name) - returns the index of a list item by its **name**.

```
>>> index = list.ItemIndexByName('VALUE')
```

ItemValueByName(name) - returns the value of a list item by its **name**. A value can be of any simple type, including None.

```
>>> value = list.ItemValueByName('VALUE')
```

ItemValueByNameDef(name, default) - returns the value of a list item by its **name**. If the element with this name is not found, it returns the **default** value.

```
>>> value = list.ItemValueByNameDef('VALUE', 5)
```

InsertItem(index, name, value) - inserts a new element with **name** and **value** into the list, at the **index** position. Returns the added item.

```
>>> item = list.InsertItem(0, 'VALUE', 5)
```

SetItem(name, value[, canadd = False]) - changes the value of a list item with **name** to a new **value**. If the additional **canadd** parameter is True and the value is not in the list, then a new value is added at the end of the list. Returns the found or new list item.

```
>>> item = list.SetItem('VALUE', 5)
```

AddItem(name, value) - adds a new element with **name** and **value** at the end of the list. Returns a new list item.

```
>>> item = list.AddItem('VALUE', 5)
```

AddItemCopy(item) - adds a copy of the **item** at the end of the list. Returns a new list item.

```
>>> item_new = list.AddItemCopy(item_old)
```

FindRow(item, end_idx, is_row_end_sign) - searches for index of the last element of a string starting from start index **start_idx**. Returns True if a string is found (one or more list items after **start_idx**). **end_idx** is a variable whose value returns the index of the last element of the string. **is_row_end_sign** is a variable that returns True if **end_idx** points to a special string separating element.

Example:

```
from plugin import *

start_idx = 0
end_idx = CreateVarParam(-1)
is_row_end_sign = CreateVarParam(False)
row_cnt = 0
while datain.FindRow(start_idx, end_idx, is_row_end_sign):
  end_idx2 = end_idx.value
  print('Row: ' + str(row_cnt))
  print('Start index: ' + str(start_idx))
  print('End index: ' + str(end_idx2))
  if is_row_end_sign.value:
    end_idx2 -= 1
  row_cnt += 1
  start_idx = end_idx.value + 1
```

NewRow() - adds a new separator element at the end of the list.

>>> list.NewRow()

Count - is a list property that contains the number of items in the list.

>>> count = list.Count

Items[index] - свойство, которое позволяет получить элемент списка по его индексу.

>>> item = list.Items[0]

List item

Each list item has two properties:

Name - the name of the element (string value).

Value - the value of the element. A value of arbitrary type, which can also be "None."

9 Built-in functions

CreateVarParam(value) - creates an object with the initial **value** to pass a variable parameter to certain functions. Returns an object.

```
>>> from plugin import *
>>> obj = CreateVarParam(-1)
```

SendData(datasource, buffer) - sends the **buffer** byte array to the **datasource** data source. The buffer type must be bytes or bytearray. **datasource** can be retrieved when calling API functions. Normally data are sent when the program executes API function code. Please note that calling a function does not guarantee delivery of data to the recipient.

```
>>> from plugin import *
```

```
>>> SendData(datasource, b'ABCD')
```

GetModuleParameter(name) - returns a module parameter value with **name**.

```
>>> from plugin import *
>>> value = GetModuleParameter('ModulePath')
```

SetModuleParameter(name, value) - sets **value** a module parameter with **name**.

```
>>> from plugin import *
>>> value = GetModuleParameter('ModulePath')
>>> SetModuleParameter('LogTitle', 'MyPlugin')
```

10 Module parameters

In this section, you can see the list of parameter identifiers with which you can access module parameters by calling the appropriate API function. The character "[w]" in a parameter description means that the value can be changed.

ModulePath - (string) the path to the folder where the module file is installed.

ModuleName - (string) the name of the module.

ApplicationFullName - (string) the name of the main application.

ModuleRegistryRoot1 - (dword) the registry branch where settings are stored:
HKEY_LOCAL_MACHINE *или* HKEY_CURRENT_USER

ModuleRegistryRoot2 - (dword) the backup registry branch.

ModuleRegistryPath - (string) the path in the registry where settings are stored.

INIFile - (string) the name of the INI file where settings are stored. If the name is specified, then settings are not stored in the registry but in a file.

INISectionPrefix - (string) if settings are stored in a file, then the parameter stores the prefix of the INI section of the settings file for this configuration and module.

DisplayFullVersion - (string) the version of the module.

IsTemporaryLoad - (bool) indicates that the module is temporarily loaded. For example, to edit settings.

LogTitle - (string) [w] the module title, which is displayed in the log file with program messages before any message from the module. It may be the same as the module description.

Description - (string) [w] the module description that is displayed in module lists in the main program.

EventsSupported - (string) [w] a list of events that a module is ready to receive and handle. Identifiers are separated by commas. By default, this list is empty. Standard events:

LOG-MESSAGE - new text in the program message log.
NEW-LOG-FILE - a new log file with data has been created.
LOG-FILE-DELETE - deletion of the old log file with data.
NEW-DATA-PACKET - a new data packet from the parser.
ERROR-WRITE-FILE - error while writing to log file.
PORT-OPEN - data source is open (started).
PORT-CLOSE - data source is closed.
CONFIG-CHANGE - configuration has been changed by the user.

USER-LOGOOF, USER-LOGON - in service mode, end or beginning of user session.
STOP-SERVICE - in service mode, service stop.