SAS® Event Stream Processing 6.2: Using the Python Interface

Getting Started **2**
- Overview 2
- Requirements 2
- Setting Up ESPPy 3

Connecting to the ESP Server **3**

Loading Projects to Python **4**

Creating Projects in Python **7**
- Using the Methods of the ESP, Project, ContinuousQuery, and Window Classes 7
- Creating and Using Templates 11

Working with Projects in Python **15**
- Overview 15
- Visualizing Models 15
- Publishing Data to Windows 16
- Using Event Generators 18
- Interacting with Windows as Pandas DataFrames 20
- Plotting Data 22
- Stopping the Event Generator 23

Visualization Programming Objects in ESPPy **24**
- Overview 24
- Getting Started 24
- Programming Objects 25
- Creating the Visuals Instance 35
- Creating Charts 37
- Event Stream Processing Model Viewer 50
- Event Stream Processing Log Viewer 51
- Laying Out Visualizations 52

Scoring an Offline SAS Cloud Analytic Services Model with ESPPy **54**
- Overview 54
- Training and Storing a CAS Model with SWAT 54

Loading an Offline Analytic Store Model in Python 60

Getting Started

Overview

ESPPy is an open-source package for Python that is available at https://github.com/sassoftware/python-esppy. With ESPPy, you can create SAS Event Stream Processing models programmatically, connect to an ESP server, and interact with projects and their components as Python objects. These objects include continuous queries, windows, events, loggers, SAS Micro Analytic Service modules, routers, and analytical algorithms. ESPPy fully integrates with JupyterLab, which supports visualizing diagrams of your ESP projects and streaming charts.

This document shows you how to use ESPPy. It steps you through two coding examples. Their Jupyter Notebooks are available with the package. The example Python code was developed using Python 3.6 and JupyterLab packaged with the Anaconda distribution.

One example demonstrates the functionality of ESPPy with the example to process stock trades. You can download the XML code and associated CSV input files from the GitHub repository for SAS Event Stream Processing Studio examples. These files are used for the example ESPPy projects described in this document.

For more information about ESPPy, see also the README file. Refer to the API Reference for ESPPy as needed.

Requirements

To use ESPPy, you must have Python 3.4+ installed on your system. ESPPy works only with SAS Event Stream Processing 5.2 or later.

To install JupyterLab, see Installing JupyterLab.

Alternatively, you can use the Anaconda platform from Continuum Analytics. Anaconda includes Python and gives you access to JupyterLab and an extensive library of other packages and environments. The Anaconda distribution is available at its download site.

To use SAS Micro Analytic Service modules with ESPPy, you must set two environment variables on the computer system where you run the ESP server:

- MAS_M2PATH is the full pathname to a required Python program, `mas2py.py`
- MAS_PYPATH is the full pathname to the desired Python executable

For example:

- `export MAS_M2PATH=/opt/sas/viya/home/SASFoundation/misc/embscoreeng/mas2py.py`
- `export MAS_PYPATH=/usr/bin/python`

For more information about SAS Micro Analytic Service, see *SAS Micro Analytic Service: Programming and Administration Guide*.

Setting Up ESPPy

To install ESPPy, follow the instructions in the project's README file.

ESPPy uses Graphviz to visualize projects as charts in JupyterLab. Download the `graphviz` program from http://www.graphviz.org/download/ and install it on your system. Even though Graphviz is required to visualize projects, ESPPy does not require Graphviz to run.

ESPPy also uses the following packages:

- Matplotlib
- ipympl
- pandas
- Requests
- image
- ws4py
- Plotly
- ipyleaflet

Connecting to the ESP Server

To load and create SAS Event Stream Processing projects in Python, you must connect to a running ESP server. For information about starting and using the ESP server, see SAS Event Stream Processing: Using the ESP Server.

Import the `esppy` module to get ESPPy working in your JupyterLab.

```
In[1]:
import esppy
```

After you start an ESP server and import the modules that you need, you can connect to a running ESP server with the `ESP` constructor. The general syntax of the constructor is as follows:

ESP('http:*host* :*port* , <*username* >, <*password* >')

Use this constructor to create an ESP server object in Python. That object connects to a running server with the name *host* and the HTTP port number *port*. If the server uses authentication, then also specify the *username* and *password*.

```
In [2]:
server = esppy.ESP('http://host :port')
server
```

Assume that your running ESP server is named **`espserver`** and the HTTP port is **`54321`**. The output from the `server` line would be as follows:

```
Out[2]:
ESP('http://espserver:54321')
```

After you connect to the ESP server, you can see the server information in the `server_info` attribute of the server object.

```
In [3]:
server.server_info
```

```
Out[3]:
{'version': '6.2',
 'engine': 'ae495037-3652-4fd8-9bb4-3701b15ab403',
 'analytics-license': True,
 'primary-server': True,
 'pubsub': 64321,
 'http': 54321}
```

You can return a list of projects on a running ESP server by project *name* or with a *filter* with the `get_projects()` method of the ESP class:

```
ESP.get_projects([name ],[filter])
```

The `get_windows()` method similarly queries the ESP server for windows on the server by *path*, window *type*, or *filter*:

```
ESP.get_windows([path ],[type],[filter ])
```

If you have not loaded projects to the server, then the `get_projects()` and `get_windows()` methods return empty dictionaries.

```
In [4]:
server.get_projects()
```

```
Out [4]:
{}
```

```
In [5]:
server.get_windows()
```

```
Out [5]:
{}
```

Loading Projects to Python

Load projects to a running ESP server as Python objects using the `load_project()` method. The general syntax is as follows:

ESP.load_project(*project* ,<*name* >, <overwrite=True|False >, <start=True|False >, <start_connectors=True|False >)

To load a project, specify one of the following:

- an XML string

- a Project object
- a local file name that contains a valid XML project definition
- a string that contains a URL that points to a valid XML project definition

Consider the Processing Trades example. With the `trades.xml` file in the current working directory, you can define a variable for the XML project definition and use the variable as an input for the `load_project()` method.

```
In [6]:
project_path = 'trades.xml'
project_xml = project_path
```

Recall that `server` was defined as the variable that references the instance of the ESP class.

```
In [7]:
trades_project = server.load_project(project_xml, name='trades_proj')
```

The `load_project()` method parses the XML in the string that is referenced by the variable `project_xml` and loads the XML definition as a project on the ESP server. This project object is referenced in Python by the variable name `trades_project` .

After the project object is created in Python and the project is loaded to the ESP server, the `get_projects()` method returns a dictionary with an entry for the project on the ESP server:

```
In [8]:
server.get_projects()
```

```
Out[8]:
{'trades_proj': Project(name='trades_proj')}
```

Note that because the `get_projects()` method retrieves information from the ESP server, the name of the project that is returned by `get_projects()` corresponds to the project name that is specified for the `name` attribute of the `project` element in the `trades.xml` file.

Querying the server for windows with the `get_windows()` method returns a dictionary with entries for each window:

```
In [9]:
server.get_windows()
```

```
Out[9]:
{'trades_proj.trades_cq.AddTraderName': JoinWindow(name='AddTraderName',
continuous_query='trades_cq',
project='trades_proj'),
'trades_proj.trades_cq.BySecurity': AggregateWindow(name='BySecurity', continuous_query='trades_cq',
project='trades_proj'),
'trades_proj.trades_cq.LargeTrades': FilterWindow(name='LargeTrades', continuous_query='trades_cq',
project='trades_proj'),
'trades_proj.trades_cq.TotalCost': ComputeWindow(name='TotalCost', continuous_query='trades_cq',
project='trades_proj'),
'trades_proj.trades_cq.Traders': SourceWindow(name='Traders', continuous_query='trades_cq',
project='trades_proj'),
'trades_proj.trades_cq.Trades': SourceWindow(name='Trades', continuous_query='trades_cq',
project='trades_proj')}
```

Because `get_windows()` is also a method of the Project class, you can alternatively use `get_windows()` as a method of the Project class to retrieve a dictionary of windows in a project on the ESP server.

```
In [10]:
trades_project.get_windows()
```

```
Out[10]:
{'AddTraderName': JoinWindow(name='AddTraderName', continuous_query='trades_cq',
project='trades_proj'),
'BySecurity': AggregateWindow(name='BySecurity', continuous_query='trades_cq',
project='trades_proj'),
'LargeTrades': FilterWindow(name='LargeTrades', continuous_query='trades_cq',
project='trades_proj'),
'TotalCost': ComputeWindow(name='TotalCost', continuous_query='trades_cq', project='trades_proj'),
'Traders': SourceWindow(name='Traders', continuous_query='trades_cq', project='trades_proj'),
'Trades': SourceWindow(name='Trades', continuous_query='trades_cq', project='trades_proj')}
```

The components of a project are stored as Python object attributes. The `queries` attribute of a project object contains the continuous query objects of the project:

```
In [11]:
trades_project.queries
```

```
Out[11]:
{'trades_cq': ContinuousQuery(name='trades_cq', project='trades_proj')}
```

The `windows` attribute of a continuous query object contains the window objects of the continuous query:

```
In [12]:
trades_project.queries['trades_cq'].windows
```

```
Out[12]:
{'Trades': SourceWindow(name='Trades', continuous_query='trades_cq', project='trades_proj'),
'Traders': SourceWindow(name='Traders', continuous_query='trades_cq', project='trades_proj'),
'LargeTrades': FilterWindow(name='LargeTrades', continuous_query='trades_cq',
project='trades_proj'),
'AddTraderName': JoinWindow(name='AddTraderName', continuous_query='trades_cq',
project='trades_proj'),
'TotalCost': ComputeWindow(name='TotalCost', continuous_query='trades_cq', project='trades_proj'),
'BySecurity': AggregateWindow(name='BySecurity', continuous_query='trades_cq',
project='trades_proj')}
```

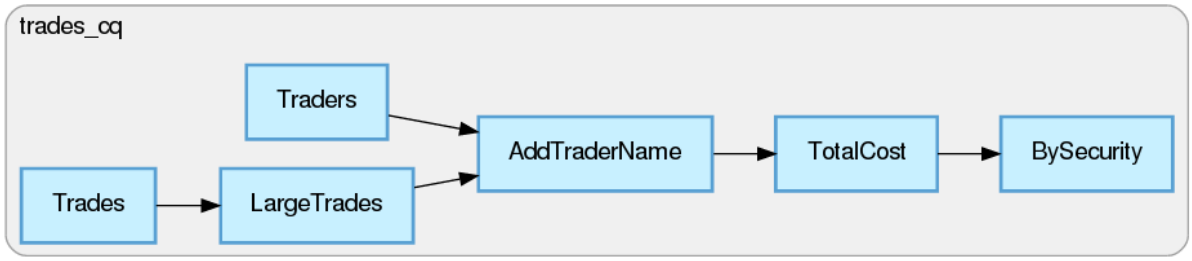
Note: The `get_projects` and `get_windows` methods create objects that reference the projects or windows on a running ESP server. Modifying the attributes of these objects affects only the objects in the Python session. Changes to these objects must be loaded to the server with the `load_project` method before the changes appear on the server.

For information about creating SAS Event Stream Processing projects in Python, see “Creating Projects in Python”.

Using Graphviz, which is imported automatically when you import ESPPy, you can produce graphs for all project, continuous query, and window objects:

```
In [13]:
trades_project.queries['trades_cq']
```

```
Out [13]:
```



You can use the `to_xml()` method to return any project object in XML form. Use the `pretty=True` argument to display the XML in the standard format. For example, running the following code returns the XML definition that corresponds to the `trades_project` object in standard XML format:

```
In [14]:
print(trades_project.to_xml(pretty=True))
```

Creating Projects in Python

Using the Methods of the ESP, Project, ContinuousQuery, and Window Classes

You can create a SAS Event Stream Processing project in Python using the methods of the ESP, Project, ContinuousQuery, and Window classes.

Create a project with the `create_project()` ESP method. Specify the name of the project that you want to create as the *project*:

ESP.create_project(*project*)

Create continuous query objects with the `add_query()` Project method. The `add_query()` method adds an empty continuous query to the project that you use to call the method. Specify the name of the continuous query as the *name*:

project.add_query(*query_name*)

Each window has its own constructor method. You create window objects with the appropriate *`Window()`* method. Specify the keyword arguments specific to the window type as individual values following the *name* of the window:

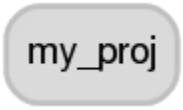
Window(*name*,<*kwargs*>)

You can add the window to an existing continuous query using the `add_window()` ContinuousQuery method.

Consider the Processing Trades example. Start by creating a project with the `create_project()` method. Remember that Graphviz, which is imported automatically when you import ESPPy, produces graphs for created objects.

```
In [15]:
my_project = server.create_project('my_proj')
my_project
```

Out [15]:



Create a continuous query within the project by using the `add_query()` method:

```
In [16]:
my_query = my_project.add_query('trades_cq')
my_query
```

Out [16]:

Create a Source window for the continuous query with the `SourceWindow` constructor.

```
In [17]:
source_trades = server.SourceWindow(name='Trades',
                                    schema=('tradeID*:string', 'security:string',
'quantity:int32',
                                            'price:double', 'traderID:int64',
'time:stamp'),
                                    index_type='rbtree')
source_trades
```

Out [17]:

Add the Source window to the continuous query with the `add_window()` method:

```
In [18]:
my_query.add_window(source_trades)
my_project
```

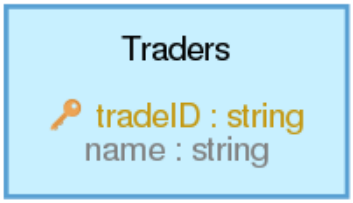
Out [18]:

The project my_proj contains the continuous query trades_cq, which contains the Source window Trades.

You can use the window-specific constructors to create the remaining windows in the Processing Trades example:

```
In [19]:
source_traders = server.SourceWindow(name='Traders', schema=('tradeID*:string',
'name:string'))
source_traders
```

```
Out [19]:
```




```
In [20]:
filter_large = server.FilterWindow(name='LargeTrades')
filter_large.set_expression('quantity >= 100')
```

```
In [21]:
join = server.JoinWindow(name='AddTraderName', type='leftouter', conditions=[['tradeID',
'tradeID']])

field_selections = [['security', '1_security'], ['quantity', '1_quantity'], ['price',
'1_price'],
                    ['traderID', '1_traderID'], ['time', '1_time'], ['name', 'r_name']]

for i in range(len(field_selections)):
    join.add_field_selection(field_selections[i][0], field_selections[i][1])
```

```
In [22]:
compute = server.ComputeWindow(name='TotalCost',
                                    schema=('tradeID*:string', 'security:string',
'quantity:int32',
                                            'price:double', 'totalCost:double',
'traderID:int64',
                                            'time:stamp', 'name:string'))
```

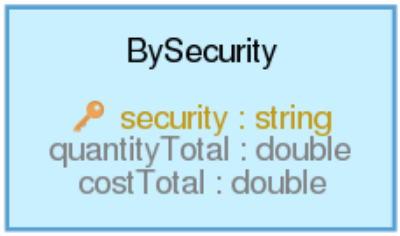
```
compute.add_field_expressions('security', 'quantity', 'price', 'price*quantity',
'traderID', 'time', 'name')
compute
```

Out[22]:


```
In [23]:
aggregate = server.AggregateWindow(name='BySecurity', schema=('security*:string',
'quantityTotal:double',
                                                              'costTotal:double'))

aggregate.add_field_expressions('ESP_aSum(quantity)', 'ESP_aSum(totalCost)')
aggregate
```

Out [23]:



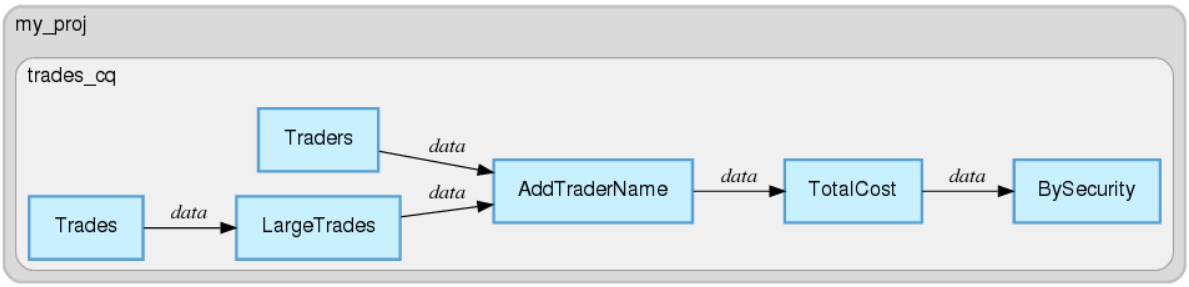
Connect the windows with the `add_target()` Window method. The `add_target()` method creates edges between windows and determines the edge roles.

```
In [24]:
source_trades.add_target(filter_large, role='data')
filter_large.add_target(join, role='left')
source_traders.add_target(join, role='right')
join.add_target(compute, role='data')
compute.add_target(aggregate, role='data')
```

Add the windows to the continuous query with the `add_window()` ContinuousQuery method:

```
In [25]:
my_query.add_windows(source_trades, source_traders, filter_large, join, compute,
aggregate)
my_project
```

```
Out [25]:
```



This project exists only locally. You now must load it to the ESP server:

```
In [26]:
server.load_project(my_project)
server.get_projects()
```

```
Out [26]:
{'my_proj': Project(name='my_proj'),
 'trades_proj': Project(name='trades_proj')}
```

As before, you can use `pretty=True` argument to the `to_xml` method to return the XML definition that corresponds to the project:

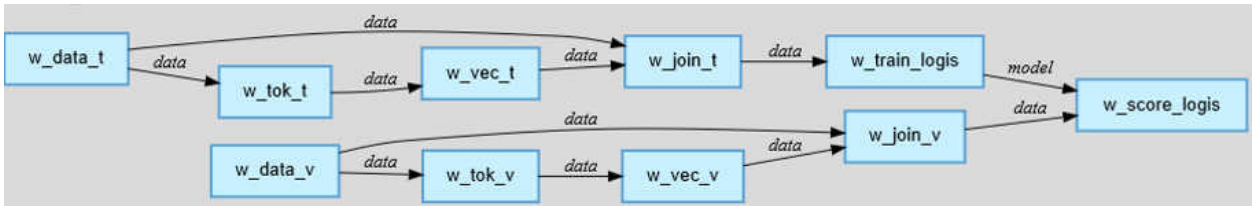
```
In [27]:
print(my_project.to_xml(pretty=True))
```

Creating and Using Templates

Sometimes, SAS Event Stream Processing projects repeatedly include sequences of windows that are connected in a specific way. ESPPy enables you to use sequences such as these as *templates*. You can use templates to simplify your Python code and to refer to these sequences with a single reference. You can use the built-in templates provided with ESPPy, or you can create your own templates.

For example, suppose that you want to perform sentiment analysis on product reviews:

Figure 1 *Sentiment Analysis Project*



This project implements two streams of input events. Here are the windows in each stream:

- A Source window (`w_data`) feeds a Calculate window (`w_tok`).
- That Calculate window feeds a second Calculate window (`w_vec`).
- The second Calculate window feeds a Join window (`w_join`).

- The Join window also receives events from the Source window.

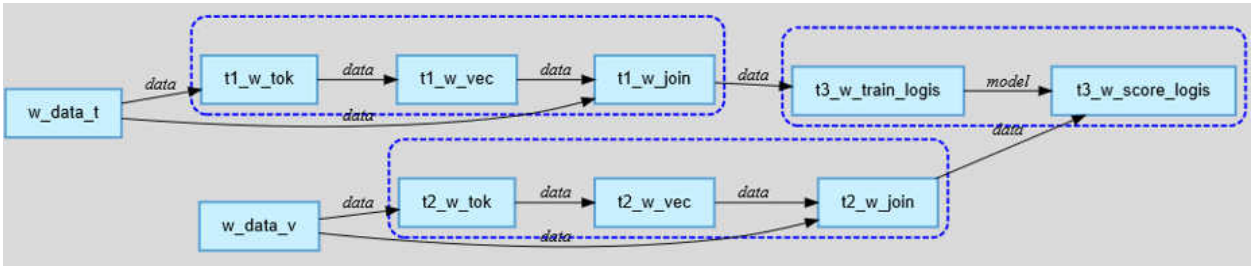
Both streams converge on a Score window, but the first stream trains the model before streaming events into that Score window.

Two sequences of windows in the model can be replaced by built-in templates provided by ESPPy.

- One sequence, which consists of a Calculate window for text tokenization, a Calculate window for text vectorization, and a Join window, occurs twice. This sequence can be referenced with the built-in TextEmbedding template.
- The other sequence consists of a Train window and a Score window for streaming logistic regression. It can be referenced with the built-in Logistic template.

The following figure depicts the same project, now showing those sequences of windows as templates:

Figure 2 *Sentiment Analysis Projects with Built-In Templates*



Here is a complete list of built-in templates provided by ESPPy:

Table 1 *Built-in Templates*

Template	Constituent Windows	Description
TextEmbedding	Two Calculate windows (`w_tok`, `w_vec`), Join window (`w_join`)	Use this template to convert document text into numeric vectors (text embedding and document embedding). The template decomposes document text into words, retrieves word embeddings for each word, and aggregates embeddings to obtain a document embedding.
Logistic	Train window (`train_logis`), Score window (`score_logis`)	Use this template to train and score a streaming logistic regression model. The template trains on a data stream with a target. Then, it makes binary predictions on another data stream.
KeypointsDetection	Calculate window (`w_resized`), Model Reader window (`w_reader`), Score window (`w_score`)	Use this template to determine keypoints on streaming images. The template resizes each incoming image to the specified width and height. It then scores them through a pre-trained keypoints detection model (in the

Template	Constituent Windows	Description
		ASTORE format). The output is the coordinates (X,Y) of keypoints on the images.
ObjectDetection	Calculate window (`w_resized`), Model Reader window (`w_reader`), Score window (`w_score`)	Use this template to find bounding boxes of objects in streaming images. The template resizes each incoming image to the specified width and height. It then scores them through a pre-trained object detection model (in the ASTORE format). The output is the bounding box locations (X, Y, W, H) of objects in the images.

To use a built-in template, create an instance and then initialize it.

```
t1 = esp.Template.TextEmbedding('text embedding')
```

After you create the instance, you can check the windows that are available within the template.

```
In [5]: t1.windows
Out[5]: {'w_join': JoinWindow(name='t1_w_join', contquery=None, project=None), 'w_tok': CalculateWindow(name='t1_w_tok', contquery=None, project=None), 'w_vec': CalculateWindow(name='t1_w_vec', contquery=None, project=None)}
```

The following code creates two instances of the TextEmbedding pre-built template and one instance of the Logistic template for a sentiment review project:

```
proj = esp.create_project('sentiment_review')

src = esp.SourceWindow(schema=('id*:int64', 'content:string', 'sentiment:string'),
                       index_type='empty', insert_only=True, autogen_key=True)

t1 = esp.Template.TextEmbedding('text embedding') #1

src.add_target(t1, role='data') #2
src.add_target(t1.windows['w_join'], role='data')

proj.windows['w_data_t'] = src
proj.add_template(t1)

src2 = esp.SourceWindow(schema=('id*:int64', 'content:string', 'sentiment:string'),
                       index_type='empty', insert_only=True, autogen_key=True)

t2 = t1.copy('t2', deep=True, internal_only=True) #3

src2.add_target(t2, role='data') #4
src2.add_target(t2.windows['w_join'], role='data')

proj.windows['w_data_v'] = src2
proj.add_template(t2)


t3 = esp.Template.Logistic('logistic regression')#5
```

```
t1.add_target(t3, role='data') #6
t2.add_target(t3.windows['w_score_logis'], role='data')
proj.add_template(t3)
```

1 Initialize an instance of the pre-built TextEmbedding template.

2 Add an edge between the first Source window (`src`) and template `t1`.

3 Create a copy of `t1` named `t2`.

4 Add an edge between the second Source window `src2` and template `t2`.

5 Initialize an instance of the pre-built Logistic template.

6 Add an edge between template `t1` and template `t3`.

ESPPy enables you to create custom template objects. For example, you can use the following code to define a template that performs postprocessing on sentiment analysis results:

```
esp = esppy.ESP('ESP_server_url')
my_template = esp.Template('my custom template') #1

comp_logis = esp.ComputeWindow("w_comp_logis",
                               schema=['id*:int64', 'sentiment:string',
                               'predicted_y:double', 'p_1:double', 'p_0:double']) #2

comp_logis.add_field_expression("tostring(tointeger(sentiment))")
comp_logis.add_field_expression("predicted_y")
comp_logis.add_field_expression("predicted_y")
comp_logis.add_field_expression("1-predicted_y") #3

fitstat_logis = esp.calculate.FitStat('fitstat_logis',
schema=('id*:int64','mceOut:double'),
                                      classLabels='0,1',
                                      windowLength=200) #4
fitstat_logis.set_inputs(inputs=('p_0:double', 'p_1:double'),
                         response=('sentiment:string'))
fitstat_logis.set_outputs(mceOut='mceOut:double')

comp_logis.add_target(fitstat_logis, role='data')#5

my_template.add_windows(comp_logis, fitstat_logis) #6

my_template.input_windows=[comp_logis]
my_template.output_windows=[fitstat_logis]
my_template.input_windows, t.output_windows #7
```

1 Create an instance of a custom template named `my_template`.

2 Define a Compute window to calculate predicted sentiment values.

3 `predicted_y` is the predicted P(sentiment = 1).

4 Define a Calculate window to apply fit statistics to those calculations.

5 Add an edge between the Compute and Calculate windows.

6 Add the Compute and Calculate windows to your custom template.

7 Set `input_windows` and `output_windows` for your custom template.

A custom template can have multiple input windows and multiple output windows. A window or template that connects to a custom template connects to the first input window that you specify in `input_windows` or `output_windows`.

For example, consider the following lines of code:

```
my_template.input_windows=[comp_logis, comp_other]
my_template.output_windows=[fitstat_logis, fitstat_other]
```

Suppose you connect `win_instance` to this template:

```
win_instance.add_target(my_template)
```

In this case, `win_instance` connects to `comp_logis`.

Similarly, consider the following line of code:

```
my_template.add_target(win_instance)
```

Here, `fitstat_logis` connects to `win_instance`.

Finally, suppose that you have set up two instances of your custom template. Consider the following line of code:

```
my_template1.add_target(my_template2)
```

This connects the first output window of the first template instance to the first input window of the second template instance.

Working with Projects in Python

Overview

After you have loaded an XML project or created a project definition, you can use different methods to visualize, process, and analyze data in a SAS Event Stream Processing model.

Visualizing Models

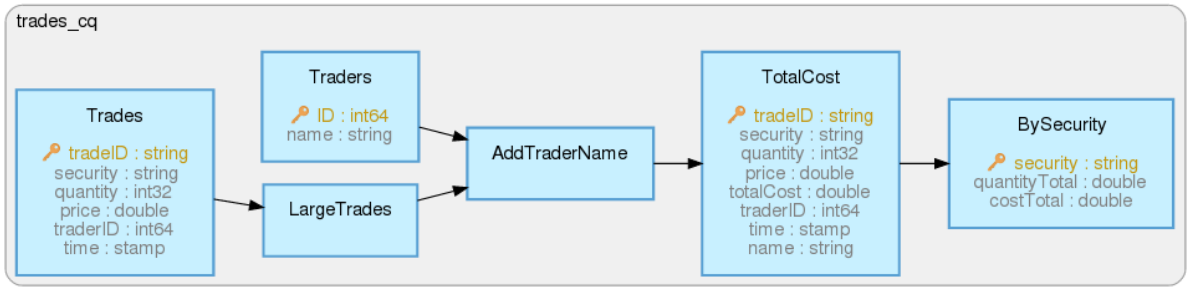
Graphviz enables you to visualize objects in a SAS Event Stream Processing model.

As you have seen when loading and creating models, requesting an object in a single cell of a Jupyter Notebook returns the Graphviz graphic visualization.

To display a graph of an object with schema, specify `schema=true` as an argument in the `to_graph()` method:

```
In [28]:
trades_project['trades_cq'].to_graph(schema=True)
```

```
Out [28]:
```



To return an object of a SAS Event Stream Processing model in XML format, use the `to_xml()` method:

```
In [29]:
print(trades_project['trades_cq']['Traders'].to_xml(pretty=True))
```

```
Out [29]:
<window-source name="Traders">
  <schema copy-keys="false">
    <fields>
      <field key="true" name="ID" type="int64" />
      <field key="false" name="name" type="string" />
    </fields>
  </schema>
</window-source>
```

Publishing Data to Windows

After loading or creating a project object, you can inject data directly into a Source window with the `publish_events()` method of the SourceWindow subclass.

***window*.publish_events** (*data* , <*args* >)

Consider the Processing Trades example data, which contains two different Source windows that read in data from two different data sources.

First, open the data in Python. Use the sample data files `trades.csv` and `traders.csv` that you downloaded from GitHub repository for SAS Event Stream Processing Studio examples. With these CSV files in the current working directory, open and read the data and assign variables to the strings.

```
In [30]:
trades_csv = open('trades.csv').read()
traders_csv = open('traders.csv').read()
print(trades_csv)
```

```
Out [30]:
i,n,TID1234321,ibm,1000,100.1,10002,08/Jul/2012:08:10:00.000000
i,n,TID1234322,sap,750,34.2,10003,08/Jul/2012:08:10:01.000345
i,n,TID1234323,ibm,1000,100.2,10004,08/Jul/2012:08:10:12.000001
i,n,TID1234324,ibm,1000,100.3,10004,08/Jul/2012:08:10:13.000002
i,n,TID1234325,ibm,1000,100.4,10004,08/Jul/2012:08:10:13.000002
i,n,TID1234326,sap,1000,34.3,10003,08/Jul/2012:08:10:13.000003
i,n,TID1234327,ibm,1000,100.3,10002,08/Jul/2012:08:10:13.000004
i,n,TID1234328,sap,1000,32,10003,08/Jul/2012:08:10:13.000005
i,n,TID1234329,ibm,1000,100,10004,08/Jul/2012:08:10:13.000006
i,n,TID1234330,sap,90,32,10003,08/Jul/2012:08:10:13.000011
i,n,TID1234331,ibm,80,100.3,10004,08/Jul/2012:08:10:13.000012
```

Note that the columns of the sample trades data correspond to the schema of the Trades Source window.

Assign variables to your Source windows:

```
In [31]:
w_trades = trades_project['trades_cq']['Trades']
w_traders = trades_project['trades_cq']['Traders']
```

Call the `publish_events()` method and specify your data source, in addition to any optional arguments such as `dateformat` for time fields and `format` for data input type:

```
In [32]:
w_trades.publish_events(trades_csv, dateformat='%d/%b/%Y:%H:%M:%S', format='csv')
w_traders.publish_events(traders_csv, format='csv')
```

The `get_events()` method returns a Pandas DataFrame whose data shows a snapshot of the events stored in the window.

```
In [33]:
w_trades.get_events()
```

```
Out [33]:
```

tradeID	security	quantity	price	traderID	time
TID1234321	ibm	1000	100.1	10002	2012-07-08 08:10:00.000000
TID1234322	sap	750	34.2	10003	2012-07-08 08:10:01.000345
TID1234323	ibm	1000	100.2	10004	2012-07-08 08:10:12.000001
TID1234324	ibm	1000	100.3	10004	2012-07-08 08:10:13.000002
TID1234325	ibm	1000	100.4	10004	2012-07-08 08:10:13.000002
TID1234326	sap	1000	34.3	10003	2012-07-08 08:10:13.000003
TID1234327	ibm	1000	100.3	10002	2012-07-08 08:10:13.000004
TID1234328	sap	1000	32.0	10003	2012-07-08 08:10:13.000005
TID1234329	ibm	1000	100.0	10004	2012-07-08 08:10:13.000006

Using Event Generators

You can load event generators to ESPPy from XML definitions or you can create them using methods of the EventGenerator class.

To load an event generator, start by reading in the XML definition as a string. Be sure to change the *hostname* and *port* in the XML before loading the event generator into Python. Also be sure to specify the publish-subscribe port in the `publish-target` element.

```
In [34]:
EventGenerator_Path = 'eventgenerator.xml'
EventGenerator_xml = open(EventGenerator_Path).read()
generator_xml = open('eventgenerator.xml').read()
print(generator_xml)
```

```
Out [34]:
<event-generator name='trades_generator' insert-only='true' autogen-key='false'>
    <publish-target>dfESP://espserver:<pubsub-port>/trades_proj/trades_cq/Trades</publish-target>
    <init>
        <value name='ID'>0</value>
    </init>
    <fields>
        <field name='tradeID'>if(equals(0,$tradeID),increment($ID),increment($tradeID))</field>
        <field name='security'>randomParm('IBM', 'SAP')</field>
        <field name='quantity'>random(80,1000)</field>
        <field
name='price'>if(equals($security,'IBM'),precision(random(100,101),1),precision(random(32,35),1))</
field>
        <field name='traderID'>if(equals($security,'IBM'),randomParm(10002,10004),10003)</field>
        <field name='time'>timeCurrent()</field>
    </fields>
</event-generator>
```

Use the `EventGenerator()` constructor to create the EventGenerator object:

```
In [35]:
trades_generator = esppy.evtgen.EventGenerator.from_xml(str(generator_xml))
trades_generator
```

```
Out [35]:
EventGenerator(name='trades_generator', publish_target='dfESP://espserver:54321/trades_proj/
trades_cq/Trades')
```

Before the event generator can be started, it must be assigned to a project session. Define the session attribute of the event generator to the project session where the event generator streams events. Then save, initialize, and start the generator with the `save()`, `initialize()`, and `start()` EventGenerator methods. Note that the `rate` parameter of the `start()` method is set to inject just 10 events per second.

```
In [36]:
trades_generator.session=trades_project.session
trades_generator.save()
trades_generator.initialize()
trades_generator.start(rate=10)
```

After you start the event generator, you can retrieve events from the Trades Source window:

```
In [37]:
w_trades2 = trades_project['trades_cq']['Trades']
w_trades2.get_events()
```

Out [37]:

tradeID	security	quantity	price	traderID	time
100	IBM	603	101.0	10002	2018-10-11 13:48:29.579938
200	IBM	146	100.0	10004	2018-10-11 13:48:39.636398
300	SAP	536	33.0	10003	2018-10-11 13:48:49.680544
110	IBM	545	100.0	10002	2018-10-11 13:48:30.587824
210	IBM	235	101.0	10002	2018-10-11 13:48:40.641984
310	IBM	815	101.0	10002	2018-10-11 13:48:50.686137
120	IBM	459	100.0	10004	2018-10-11 13:48:31.592107
220	IBM	479	101.0	10002	2018-10-11 13:48:41.647340
320	IBM	314	100.0	10002	2018-10-11 13:48:51.691383

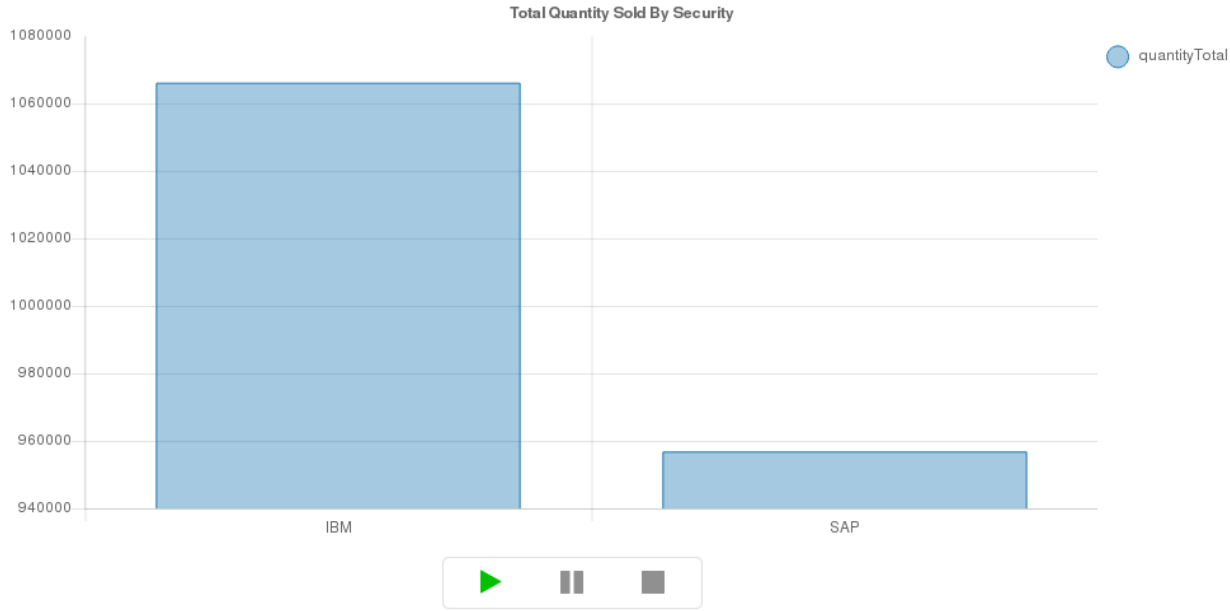
You can import StreamingChart in order to create a streaming bar chart from events as they stream through the Aggregate window BySecurity.

```
In [38]:
from esppy.plotting import StreamingChart

w_BySecurity=trades_project['trades_cq']['BySecurity']
streamChart=w_BySecurity.streaming_bar('security', 'quantityTotal', title='Total
Quantity Sold By Security')#1
streamChart
```

1 When you specify a streaming chart, it creates an independent subscriber to the associated window. Thus, you do not need to subscribe to that window on a separate line of code beforehand.

Out [38]:



For information about other streaming charts, including line, scatter, and pie charts, see https://sassoftware.github.io/python-esppy/api.html?highlight=streaming%20charts#streaming-charts.

Interacting with Windows as Pandas DataFrames

The `subscribe()` method of the Window class creates a Pandas DataFrame whose columns and index mirror the schema of the window. This DataFrame is stored as the `data` attribute of the Window object. After the DataFrame for a window has been created and stored, you can access any DataFrame attribute and call any DataFrame method on the Window object as if it were a DataFrame.

After you use the `subscribe()` Window method to create a subscriber to a window, returning a Window object displays a snapshot of the events in the window's `data` attribute.

```
In [39]:
w_trades2.subscribe()
w_trades2
```

```
Out [39]:
```

tradeID	security	quantity	price	traderID	time
111	IBM	133	100.0	10004	2018-04-23 17:08:26
112	IBM	252	101.0	10002	2018-04-23 17:08:26
113	IBM	821	100.0	10004	2018-04-23 17:08:26
114	IBM	449	101.0	10002	2018-04-23 17:08:26
115	SAP	498	32.0	10003	2018-04-23 17:08:26
116	SAP	533	35.0	10003	2018-04-23 17:08:26
117	SAP	280	32.0	10003	2018-04-23 17:08:26
118	SAP	217	34.0	10003	2018-04-23 17:08:26
119	SAP	667	32.0	10003	2018-04-23 17:08:26
120	IBM	484	101.0	10002	2018-04-23 17:08:26
121	SAP	242	33.0	10003	2018-04-23 17:08:27
122	IBM	151	101.0	10004	2018-04-23 17:08:27
123	SAP	410	32.0	10003	2018-04-23 17:08:27
124	SAP	249	33.0	10003	2018-04-23 17:08:27

Use methods of the DataFrame class to interact with your windows as DataFrames.

```
In [40]:
w_trades2.info()
```

```
Out [40]:
<class 'pandas.core.frame.DataFrame'>
Index: 7142 entries, 4671 to 11913
Data columns (total 5 columns):
security    7142 non-null object
quantity    7142 non-null int64
price       7142 non-null float64
traderID    7142 non-null int64
time        7142 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(1), int64(2), object(1)
memory usage: 306.3+ KB
```

```
In [41]:
w_trades2.head()
```

```
Out [41]:
```

tradeID	security	quantity	price	traderID	time
4761	IBM	122	101.0	10002	2018-10-11 14:44:52.930613
4762	IBM	930	101.0	10002	2018-10-11 14:44:52.930860
4763	SAP	698	33.0	10003	2018-10-11 14:44:52.930977
4764	IBM	235	100.0	10002	2018-10-11 14:44:52.931083
4765	SAP	558	35.0	10003	2018-10-11 14:44:52.931196

```
In [42]:
w_trades2.describe()
```

Out [42]:

	quantity	price	traderID
count	140.000000	140.000000	140.000000
mean	545.478571	66.992857	10002.957143
std	273.829450	33.593100	0.708341
min	81.000000	32.000000	10002.000000
25%	286.000000	34.000000	10002.000000
50%	522.500000	67.500000	10003.000000
75%	770.000000	100.000000	10003.000000
max	995.000000	101.000000	10004.000000

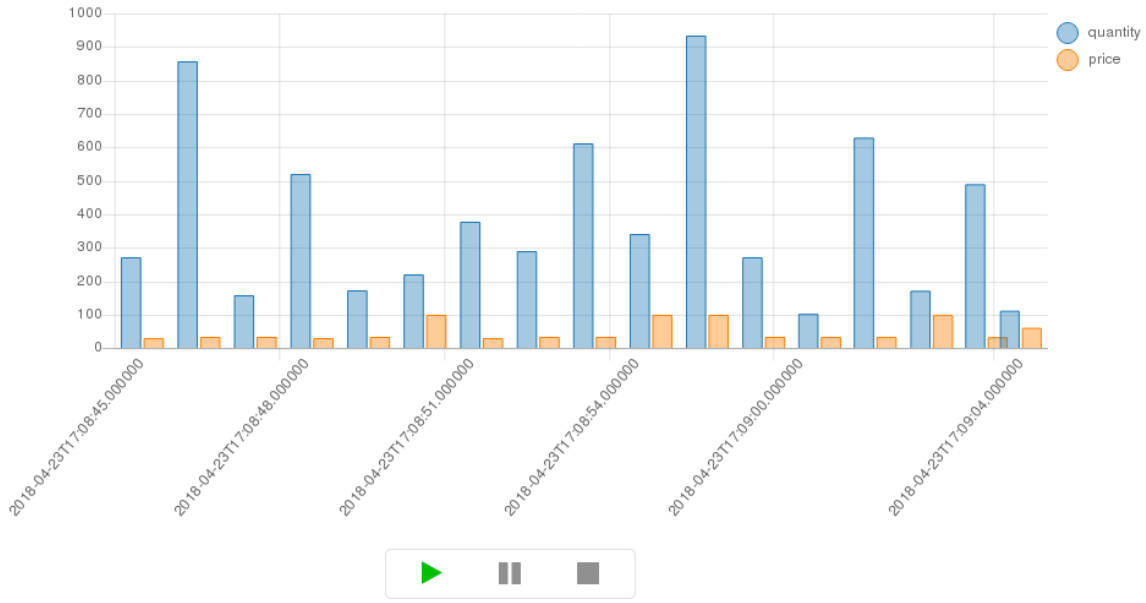
For the full Pandas API reference, see https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

Plotting Data

You can plot data in ESPPy as streaming charts.

```
In [43]:
w_trades2.streaming_bar(x='time', y=['quantity', 'price'])
```

Out [43]:

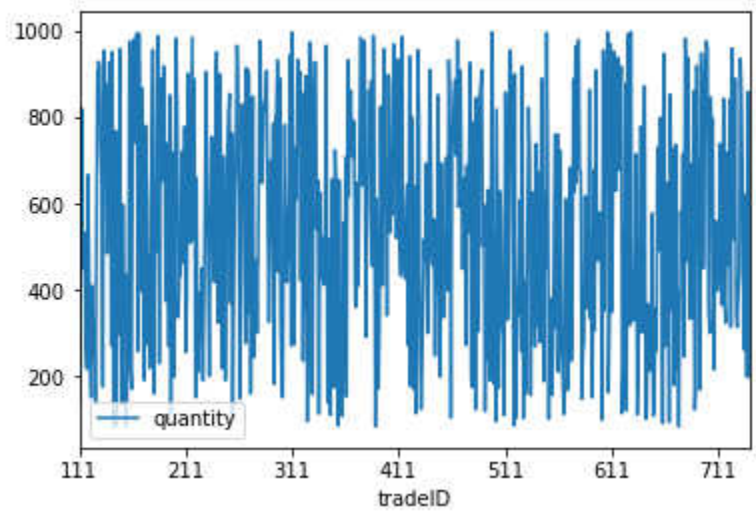


You can also use the plotting methods of Pandas to plot snapshots of data. To plot your DataFrame, use `matplotlib`.

```
In [44]:
%matplotlib inline
w_trades2.plot(y='quantity')
```

```
Out [44]:
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcebd93c860>
```



Stopping the Event Generator

When you want to stop the generator, use the `stop()` method. After the event generator is stopped, use the `delete()` method to remove the generator from the server:

```
In [45]:
trades_generator.stop()
trades_generator.delete()
```

Visualization Programming Objects in ESPPy

Overview

ESPPy provides the capability to visualize ESP models in action within a Jupyter Lab environment. You can create various charts, an ESP model viewer, and an ESP log viewer using the following programming objects:

- ServerConnection
- EventCollection
- EventStream
- Publisher
- Stats
- Log

You can use these objects to build the following visualizations:

- Bar Charts
- Line Charts
- Time Series Graphs
- Bubble Plots
- Pie Charts
- Maps
- Gauges
- Compasses
- Tables
- Event Stream Processing Model Viewer on page 50
- Event Stream Processing Log Viewer

Getting Started

To use the visualization objects, follow the instructions in ESPPy's README file.

Programming Objects

ServerConnection Object

Create a ServerConnection object to create the data sources that feed visualizations. The ServerConnection is a persistent connection to the ESP server. Create an instance of this object through an ESP instance:

```
esp = esppy.ESP("http://myserver:33000")
conn = esp.createServerConnection()
```

The visualization API uses this connection to communicate with the ESP server. Also, the connection is used to monitor the health of the server. When the server goes down while the connection is active, the API waits for the server to return. When the server comes back, all data sources are reconnected automatically.

ServerConnection Methods

getEventCollection

Create and return an EventCollection object.

getEventCollection(*path*,*keywordparameters*)

Table 2 *Parameter for getEventCollection*

Parameter	Description
path	Specify the path to the window from which to obtain the event. Use the format */project/contquery/window*.

Table 3 *Keyword Parameters for getEventCollection*

Parameter	Description
pagesize	Specify the page size for the collection. The default value is 50.
filter	Specify a functional filter for the collection.
interval	Specify the time, in milliseconds, for the server to wait before delivering any events that occurred. If not specified, the interval defaults to 1 second.

Example Code 1 *Examples*

```
brokerAlerts = conn.getEventCollection("sec/cq/brokerAlertsAggr")
venueAlerts = conn.getEventCollection("sec/cq/
venueAlertsAggr",filter="in($city,'raleigh'))
```

getEventStream

Create and return an EventStream object. An event stream is similar to a UNIX tail of an event stream processing window.

getEventStream(*pathkeywordparameters*)

Table 4 *Parameter for getEventStream*

Parameter	Description
path	Specify the path to the window from which to obtain the event. Use the format */project/contquery/window*.

Table 5 *Keyword Parameters for getEventStream*

Parameter	Description
maxevents	Specify the maximum number of events to store in the collection. The default value is 100.
interval	Specify the time, in milliseconds, for the server to wait before delivering any events that occurred. If not specified, the interval defaults to 1 second.

Example Code 2 *Example*

```
rates = conn.getEventStream("primary/cq/counter",maxevents=20)
```

getPublisher

Create and return a publisher object.

getPublisher(*path*)

Table 6 *Required Parameter for getPublisher*

Parameter	Description
path	Specify the path to the Source window. Use the format */project/contquery/window*.

Example Code 3 *Example*

```
publisher = conn.getPublisher("primary/cq/rawTrades")
```

publishURL

Publish data from a URL into the ESP server. The URL is sent in a request to the server. The server pulls the data from the URL and injects it into the model.

publishURL(*path*, *url*, *keywordparameters*)

Table 7 Required Parameters for publishURL

Parameter	Description
path	Specify the path to the Source window. Use the format */project/contquery/window*.
url	Specify the URL from which to read event data. Event data can be in CSV, XML, JSON, or binary format.

Table 8 Keyword Parameters for publishURL

Parameter	Description
blocksize	Specify the event block size to use when injecting events into the ESP server. The default value is 1.
wait	Specify **true** or **false** to indicate whether to wait until publishing is complete in the server before the function returns a value. The default value is **false**.

Example Code 4 Example

```
conn.publishUrl("p/cq/s","http://myserver:9999/events.csv",wait=True)
```

publishDataFrame

Publish a pandas data frame to the ESP server.

publishDataFrame(*path*, *dataframe*, *keywordparameters*)

Table 9 Required Parameters for publishDataFrame

Parameter	Description
path	Specify the path to the Source window. Use the format */project/contquery/window*.
dataframe	Specify the data frame to publish.

Table 10 Keyword Parameters for publishDataFrame

Parameter	Description
size	Specify the number of events to send in each publish request. The default value is 100.
blocksize	Specify the event block size to use when injecting events into the ESP server. The default value is 1.

Example Code 5 *Examples*

```
df = pandas.read_csv("trades.csv",header=0)
conn.publishDataFrame("primary/cq/rawTrades",df)
```

publishList

Publish a Python list to the ESP server.

publishList(*path*, *list*, *keywordparameters*)

Table 11 *Required Parameters for publishList*

Parameter	Description
path	Specify the path to the Source window. Use the format */project/contquery/window*.
list	Specify the Python list to publish.

Table 12 *Keyword Parameters for publishList*

Parameter	Description
size	Specify the number of events to send in each publish request. The default value is 100.
blocksize	Specify the event block size to use when injecting events into the ESP server. The default value is 1.

Example Code 6 *Examples*

```
l = [{"id":"1","price":10},{"id":"2","price":20},{"id":"3","price":30}]
conn.publishList("p/cq/s",l)
```

getStats

Returns the Stats object for the connection

getStats()

getLog

Returns the log object for the connection

getLog()

EventCollection Object

An EventCollection is a view into a stateful window. When an EventCollection is created, it sets a page size. This size determines the maximum number of events that are sent from the server to the client. In addition, the server sets up an internal publish/subscribe instance to receive events for the window. When an event is not in the current page, that event is ignored. Otherwise, the event is delivered to the client.

Create an EventCollection through the ServerConnection object:

```
alerts = conn.getEventCollection("secondary/cq/brokerAlertsAggr")
```

The Event Collection manages itself and delivers change events to its delegates. To receive collection change notifications, you must register a delegate that implements the `dataChanged` method:

```
class MyDelegate(object):
    def dataChanged(self,collection,data,clear):
        print("data changed: " + str(data))

alerts.addDelegate(MyDelegate())
```

The `dataChanged` method receives the collection that changed along with the new data in the `data` parameter.

EventCollection Methods

addDelegate

Add a delegate to receive collection change notifications.

addDelegate(*delegate*)

Table 13 *Required Parameter for addDelegate*

Parameter	Description
delegate	Specify the object to receive change notifications.

removeDelegate

Remove a delegate from the collection.

removeDelegate(*delegate*)

Table 14 *Required Parameter for removeDelegate*

Parameter	Description
delegate	Specify the object to remove from the collection.

getData

Return the collection of events. The data is a dictionary of objects that represent events sorted by event key.

getData()

setFilter

Set a functional filter to apply to events in the window.

getData(*filter*)

Table 15 *Required Parameter for setFilter*

Parameter	Description
filter	Specify a functional filter.

Example Code 7 *Example*

```
coll.setFilter("gt($price,200)")
```

load

Load the current page of events from the ESP server.

load()

first

Load the first page of events from the ESP server

first()

last

Load the last page of events from the ESP server.

first()

next

Load the next page of events from the ESP server.

next()

prev

Load the previous page of events from the ESP server.

prev()

EventStream Object

An EventStream is a flow of events that is similar to a UNIX tail. When an event occurs in an ESP model in the server, it is placed into the EventStream. Clients can read this stream and view the events as they occur. When a window produces a very large number of events, you can throttle the number of events injected into the stream by specifying `maxevents` and `interval`.

The `maxevents` keyword parameter limits the number of events that get put into the stream at any one time. The `interval` keyword parameter specifies a time, in milliseconds, that the ESP server waits before putting events into the stream.

Suppose that you set `maxevents` to 1000, and `interval` to 1000 ms (or 1 second). The ESP server collects events from the publish/subscribe interface for 1 second. When more than `maxevents` events occur in that interval, the oldest events are dropped and are not put into the stream. At the end of 1 second, the server puts all of the events that it has into the stream. For clients with heavy processing

duties (for example, those that render graphs), this enables the client to limit the number of events that it must process.

When delete events are of no interest, specify `ignore_delete=True` when you create the EventStream. This discards any delete events received by the stream.

Create an EventStream through the ServerConnection object:

```
rates = conn.getEventStream("primary/cq/counter",maxevents=20)
```

The EventStream delivers change events to its delegates. If you want stream change notifications, you must register a delegate that implements the `dataChanged` method:

```
class MyDelegate(object):
    def dataChanged(self,stream,data,clear):
        print("data changed: " + str(data))

rates.addDelegate(MyDelegate())
```

The `dataChanged` method receives the stream that changed along with the new data in the `data` parameter.

EventStream Methods

addDelegate

Add a delegate to receive collection change notifications.

addDelegate(*delegate*)

Table 16 *Required Parameter for addDelegate*

Parameter	Description
delegate	Specify the object to receive collection change notifications.

removeDelegate

Remove a delegate from the collection.

removeDelegate(*delegate*)

Table 17 *Required Parameter for removeDelegate*

Parameter	Description
delegate	Specify the object to remove.

getData

Return the collection of events. The data is an array of objects that represent the events.

getData()

setFilter

Set a functional filter to use on events in the window.

setFilter(*filter*)

Table 18 *Required Parameter for setFilter*

Parameter	Description
filter	Specify the functional filter.

Example Code 8 *Example*

```
coll.setFilter("gt($price,200)")
```

Publisher Object

The Publisher object enables you to publish events into an ESP source window. You can add data to the send queue either by one or more sequences of `begin()`, `set()`, `end()` calls. Alternatively, you can use the `add()` method that adds an object to the send queue directly. The publisher stores the objects to be published until the `publish()` method is called.

Publisher Methods

begin

Initialize the current data.

begin()

set

Set a field value in the current data.

set(*name*, *value*)

Table 19 *Required Parameters for set*

Parameter	Description
name	Specify the name of the field to set.
value	Specify the value of the field.

end

Close input to the current data. Then add the data to the publish list.

end()

add

Add data to the publish list.

add(*data*)

Table 20 *Required Parameter to add*

Parameter	Description
data	Specify the data object to add to the publish list.

publish

Publish the publish list into the Source window.

publish()

Stats Object

The Stats object enables you to monitor ESP server statistics such as memory usage and CPU usage on a per window basis.

To receive notifications when the data changes, set up a delegate object that implements the `handleStats(self,stats)` method:

```
class MyDelegate(object):
    def handleStats(self,stats):
        print("stats changed: " + str(stats.getData()))

conn.getStats().addDelegate(MyDelegate())
```

The Stats array contains CPU and window event count information for any window that meets one or both of the following criteria:

- CPU usage is greater than the minimum value.
- The window event count is greater than 0. (Windows with an index of `pi_EMPTY` always have an event count of 0.)

The memory information includes values for system, virtual, and resident memory currently consumed in the ESP server.

Stats Methods

addDelegate

Add a delegate to receive Stats change notifications.

addDelegate(*delegate*)

Table 21 Required Parameter

Parameter	Description
delegate	Specify the object to receive change notifications.

setOpts

Set a Stats reporting option.

setOpts(*keywordparameter*)

The *keywordparameter* can be any one of the following:

- **interval** - the interval, in seconds, at which the server sends data to the client.
- **cpu** - the minimum CPU usage, in percentage, which is reported. The default value is 5.
- **memory** - specify **true** or **false** to determine whether to report memory usage information (defaults to **true**).
- **counts** - specify **true** or **false** to determine whether to report window event counts (defaults to **false**).
- **config** - specify **true** or **false** to determine whether to report ESP server configuration information (defaults to **false**).

getData

Return the Stats information.

getData()

The returned Stats data would look something like this:

```
[
    {
        'project': 'primary',
        'contquery': 'cq',
        'window': 'transform',
        'cpu': 100.598,
        'interval': 1101676.0,
        'count': 0,
        '_key_': 'primary.cq.transform'
    }, {
        'project': 'primary',
        'contquery': 'cq',
        'window': 'rawTrades',
        'cpu': 31.9233,
        'interval': 1101676.0,
        'count': 0,
        '_key_': 'primary.cq.rawTrades'
    }]
```

getMemoryData

Return memory information.

getMemoryData()

The returned memory data would look something like this:

```
{
    'system': 386696,
    'virtual': 1452,
    'resident': 241
}
```

Log Object

The Log object enables you to monitor ESP server logs. In order to receive notifications when the Stats data changes, you must add a delegate object that implements the `handleLog(log,message)` method:

```
class MyDelegate(object):
    def handleLog(self,log,message):
        print("got a log message: " + message)

conn.getLog().addDelegate(MyDelegate())
```

The message parameter is string containing the text of a log message.

Log Method

addDelegate

Add a delegate to receive log messages.

addDelegate(*delegate*)

Table 22 *Required Parameter for addDelegate*

Parameter	Description
delegate	Specify the object to receive log messages.

Creating the Visuals Instance

You need to import both `esppy` and `Visuals` in order to create the graphics.

```
import esppy
from esppy.espapi.visuals import Visuals
```

The Visuals object is used to create all the charts, tables, and viewers. The keyword parameters that you can specify when creating the Visuals instance are:

- `colormap` - the color theme

- `colors` - a list of colors in string format (for example, `["#89cff0","#0080ff","#f0bd27","#ff684c","#e03531"]`)
- `border` - the border around each graph. Use standard CSS syntax to set this property (for example, `border="1px solid blue"`).
- `margin` - the margin, in pixels, within each graph (defaults to "0px"). The value should be a string with a numeric value followed by `'px'` (for example, `margin="10px"`).

You can specify a color theme for the instance upon creation. These colors are used to render any of the graphics created by the instance.

Table 23 *Color Themes*

Theme	Elements
Plotly color scales	■ Blackbody ■ Bluered ■ Blues ■ Earth ■ Electric ■ Greens ■ Greys ■ Hot ■ Jet ■ Picnic ■ Portland ■ Rainbow ■ RdBu ■ Reds ■ Viridis ■ YlGnBu ■ YlOrRd
Matplotlib color maps	Refer to the documentation.
SAS Color Themes	■ sas_base ■ sas_corporate ■ sas_dark ■ sas_highcontrast ■ sas_ignite ■ sas_inspire ■ sas_light ■ sas_marine ■ sas_midnight ■ sas_opal

Theme	Elements
	■ sas_sail ■ sas_snow ■ sas_umstead ■ sas_hcb

The call to create the Visuals instance looks something like this:

```
visuals = Visuals(colormap="sas_corporate",border="2px solid blue",margin=5)
```

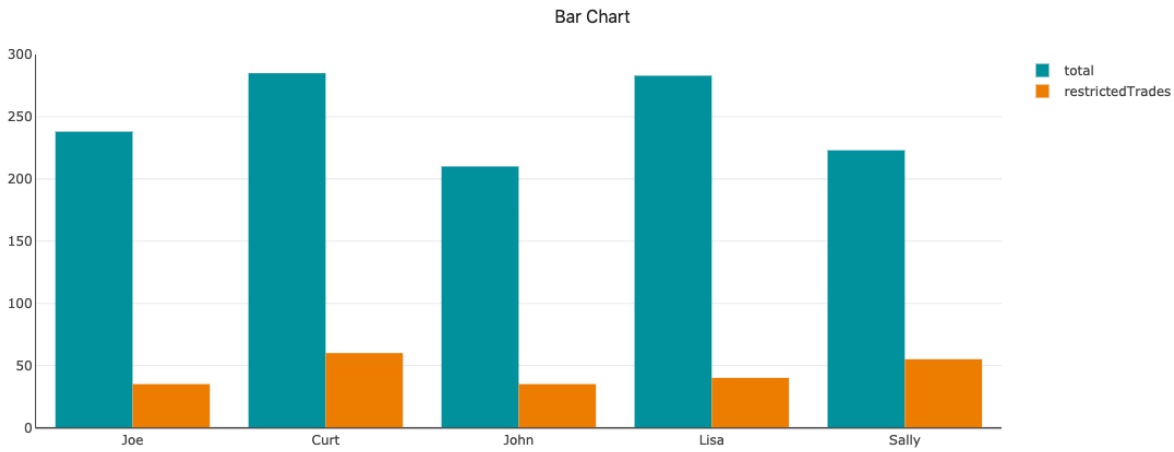
Creating Charts

Overview

After you have an active server connection active and you have created your Visuals instance, you can include charts in your JupyterLab. Invoke the appropriate chart creation method, and then use the name of the chart when and where you want to display it:

Bar Charts

Use a bar chart to plot multiple numeric variables on X and Y axes.



Use code like the following to create a bar chart:

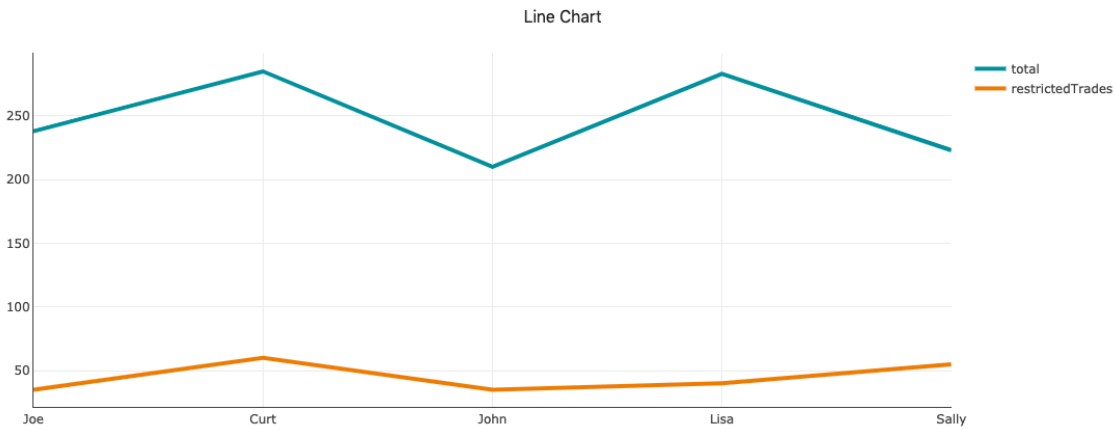
```
violationChart = visuals.createBarChart(alerts,y=["total","restrictedTrades"],
                        title="Violation Alerts by Broker")
```

Table 24 *Parameters for Bar Charts*

Parameter	Description
datasource	Specify the event collection or event stream that feeds the bar chart.
x	Specify an array of classification field values to display on the X axis. If you do not specify a value for x, event key values are used.
y	Specify an array of numeric field values to display on the Y axis (or X axis, depending on the value of orientation).
orientation	Specify the orientation of the chart. Valid values are **vertical**(default) and **horizontal**.
title	Specify a title for the chart.

Line Charts

Use a line chart to plot multiple numeric variables on the Y axis.



Use code like the following to create a line chart:

```
rateChart =
visuals.createLineChart(eventRate,y=["totalRate","intervalRate"],title="Event Rate")
```

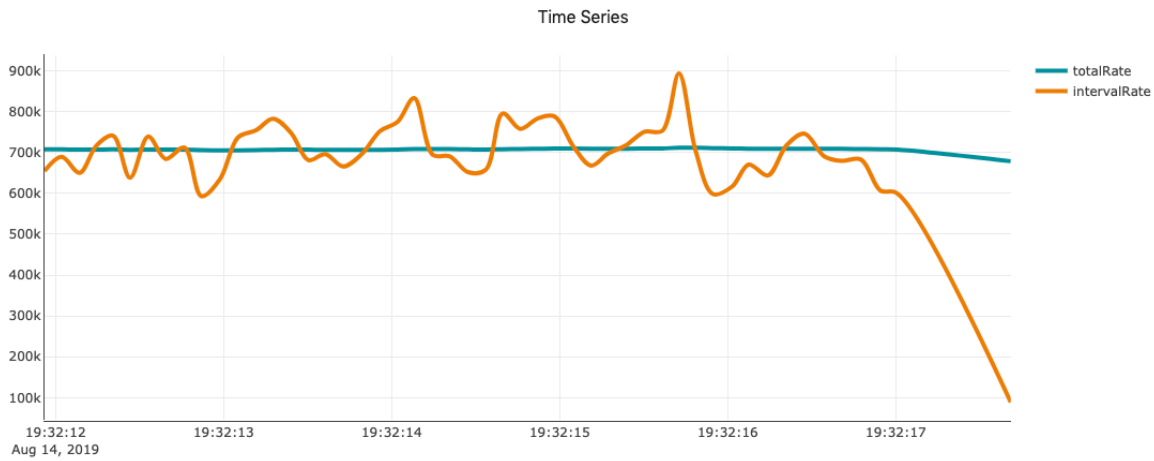
Table 25 *Parameters for Line Charts*

Parameter	Description
datasource	Specify the event collection or event stream that feeds the line chart.
x	Specify an array of classification field values to display on the X axis. If you do not specify a value for x, event key values are used.

Parameter	Description
`y`	Specify an array of numeric field values to display on the Y axis.
`line_width`	Specify the width of lines in the chart, in pixels. The default value is 2.
`curved`	Specify whether the lines are curved. Valid values are **`true`** or **`false`** (lines are straight).
`fill`	Specify whether the lines are filled underneath. Valid values are **`true`** or **`false`**.
`title`	Specify the title of the chart.

Time Series Graphs

Specify a time series graph to plot multiple numeric variables on the Y axis with a date or time variable on the X axis.



To use this graph, you must specify a field value that is either a `date` or `timestamp`. Use code like this to create a time series graph:

```
rateChart = visuals.createTimeSeries(eventRate,time="_timestamp",
                    y=["totalRate","intervalRate"],title="Event Rate")
```

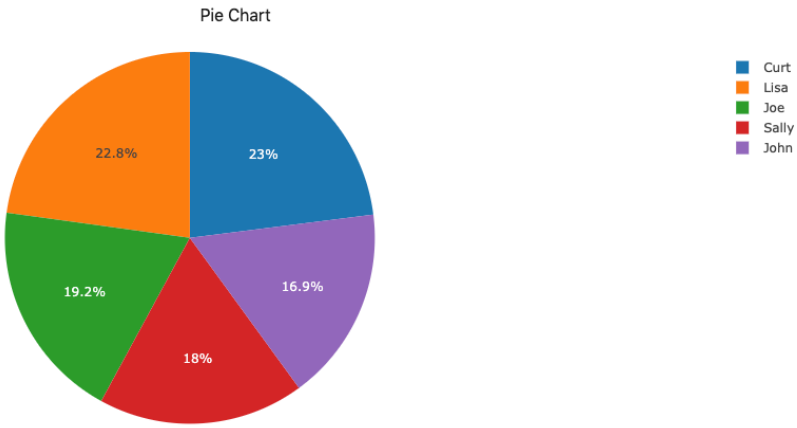
Table 26 *Parameters for Time Series Graphs*

Parameter	Description
`datasource`	Specify the event collection or event stream that feeds the time series chart.
`time`	Specify a field value of type `date` or `timestamp`.

Parameter	Description
`y`	Specify an array of numeric field values to display on the Y axis.
`line_width`	Specify the width of lines in the chart, in pixels. The default value is 2.
`curved`	Specify whether the lines are curved. Valid values are **`true`** or **`false`** (lines are straight).
`fill`	Specify whether the lines are filled underneath. Valid values are **`true`** or **`false`**.
`title`	Specify the title of the chart.

Pie Charts

Use a pie chart to represent a numeric value as a slice of the pie.



Use code like the following to create a pie chart:

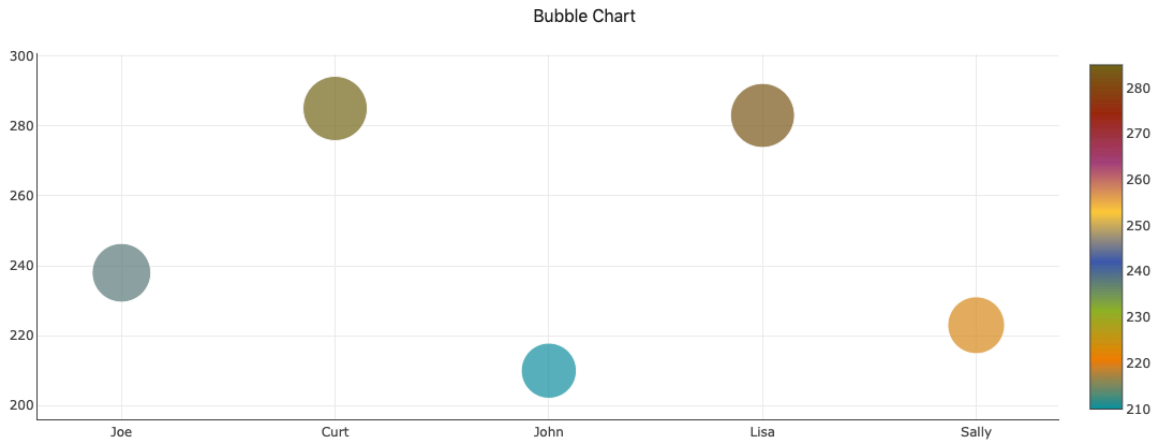
```
brokerChart = visuals.createPieChart(brokerAlerts,value="total",title="Broker Alerts")
```

Table 27 *Parameters for Pie Charts*

Parameter	Description
`datasource`	Specify the event collection or event stream that feeds the pie chart.
`value`	Specify the field value to display in the pie.
`title`	Specify the title of the chart.

Bubble Plots

Use a bubble plot to represent multiple numeric values as y, size, and color visualizations.



Use code like the following to create a bubble plot:

```
brokerChart =
visuals.createBubbleChart(brokerAlerts,y="restrictedTrades",size="total",color="frontR
unningSell",title="Broker Alerts")
```

Table 28 *Parameters for Bubble Plots*

Parameter	Description
datasource	Specify the event collection or event stream that feeds the bubble plot.
`x`	Specify an array of classification field values to display on the X axis. If you do not specify a value for `x`, event key values are used.
`y`	Specify the field values to display on the Y axis.
`size`	Specify the field value to use for the size of each bubble.
`color`	Specify the field value to use for the color of each bubble.
`title`	Specify the title of the chart.

Maps

The map chart uses the Jupyter widget `ipyleaflet` to display a map that is overlaid with markers that represent event data. You can display circles or polygons on the map.

To add a circle, you must call the `addCircles` method with an EventCollection object that contains events with the circle data. Use the following keyword parameters:

- `lat` - the field representing the latitude of the circle
- `lon` - the field representing the longitude of the circle
- `radius` - the field representing the radius of the circle
- `text` - the field that contains text to display when the user clicks on the circle

To add a polygon, you must call the `addPolygons` method with an EventCollection object that contains events with the shape data. Use the following keyword parameters:

- `coords` - the field that contains a space-separated list of latitude and longitude points in the polygon
- `text` - the field that contains text to display when the user clicks on the polygon
- `order` - the sequencing of the points. When **`lat_lon`** (default), the points are in latitude,longitude order. When **`lon_lat`**, the points are in longitude,latitude order.

Use code like the following to create a map chart:

```
venueMap = visuals.createMap(venueAlerts,lat="lat",lon="lon",size="count",
                                  color="count",title="Venue Alerts")
```

Table 29 *Parameters for Map Charts*

Parameter	Description
datasource	Specify an event collection or event stream to feed the map chart.
`lat`	Specify the field value that contains the latitude.
`lon`	Specify the field value that contains the longitude.
`size`	Specify either the field value to use to derive the size of each marker or a numeric value for fixed size markers.
`color`	Specify either the field value to use to derive the color of each marker or a fixed color.

Parameter	Description
`colormap`	Specify the color theme to use to color markers. Use this or `colors`, but not both.
`colors`	Specify a list of colors to use to color markers (for example, `["#89cff0","#0080ff","#f0bd27","#ff684c","#e03531"]`. Use this or `colormap`, but not both.
`color_range`	Specify the start and end values against which the data values are compared to derive a color. When this is not set, the minimum and maximum data values are used as the range.
`popup`	Specify the field values to display when the user clicks on a marker.
`marker_border`	Specify **`true`** or **`false`** to determine whether markers have borders.
`marker_opacity`	Specify a value between 0 and 1 to determine marker opacity.
`zoom`	Specify a value between 1 and 18 for the initial zoom value for the map. The default value is 12.
`center`	Specify a (`lat`,`lon`) pair to use as the center of the map. The default value is **`(0,0)`**.
`tracking`	Specify **`true`** or **`false`** to determine whether the map moves such that the initial marker is always in the center.
`circle_border_width`	Specify a numeric value for the width of circle borders, in pixels. The default value is 1.
`circle_border_color`	Specify the color to use for circle borders. The default value is **`black`**.
`circle_fill_color`	Specify the color to use for circle interiors. The default value is **`white`**.
`circle_fill_opacity`	Specify a numeric value for the opacity of circle interiors. The default value is .2.
`poly_border_width`	Specify a numeric value for the width of polygon borders, in pixels. The default value is 1.
`poly_border_color`	Specify the color to use for polygon borders. The default value is **`black`**.
`poly_fill_color`	Specify the color to use for polygon interiors. The default value is **`white`**.
`poly_fill_opacity`	Specify a numeric value for the opacity of polygon interiors. The default value is .2.

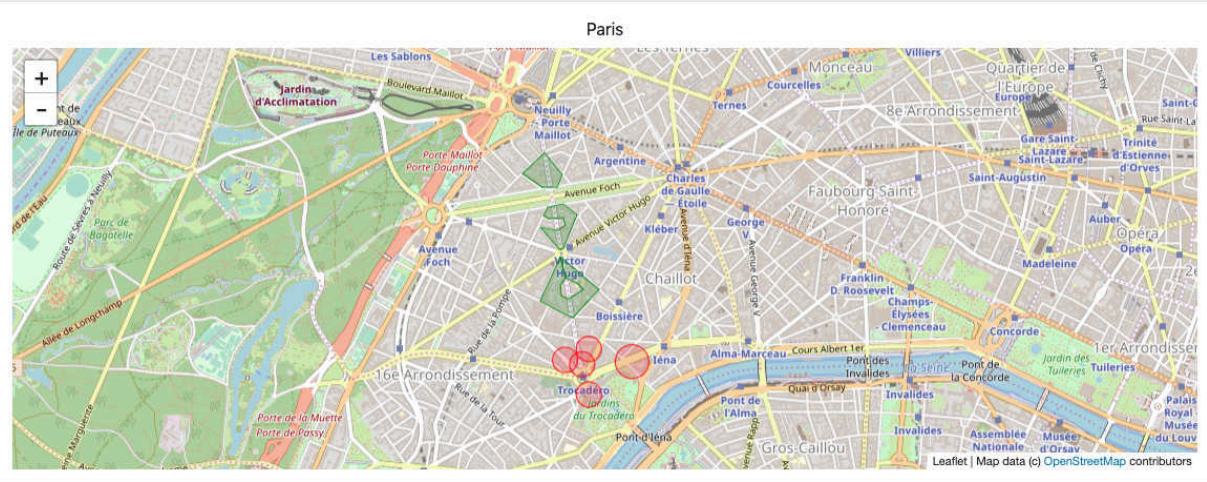
The following code generates a map of Paris and adds custom shapes to it:

```
paris =
visuals.createMap(tracker,lat="GPS_latitude",lon="GPS_longitude",size=10,color="speed"
,color_range=(0,50),

title="Paris",popup=["vehicle","speed"],marker_border=False,

colors=["#e03531","#ff684c","#f0bd27","#51b364","#8ace7e"],
                            zoom=15,tracking=True,center=(48.875,2.287583))

paris.addCircles(circles,lat="POI_y",lon="POI_x",radius="POI_radius",text="POI_desc")
paris.addPolygons(polygons,coords="poly_data",text="poly_desc",order="lon_lat")
```



Gauges

Use gauges to display a single numeric value in a graphic divided into segments.

For each event, a gauge is displayed that shows the specified numeric value of the event and a pointer that shows where that value lies in the specified range. The key value of the event is displayed at the top of the gauge with the current gauge value.

When you specify a single gauge color, it applies to the leftmost segment. The remaining segments are colored with an increasingly darker shade of the specified color. Alternatively, you can specify a color palette that is applied in the segments left to right.

Use code like this to create gauges:

```
colors = ["#8ace7e","#51b364","#f0bd27","#ff684c","#e03531"]

gauge =
visuals.createGauge(broker,segments=5,value="total",size=300,shape=80,range=(0,1000),
        columns=3,colors=colors,label="Total Alerts")
```

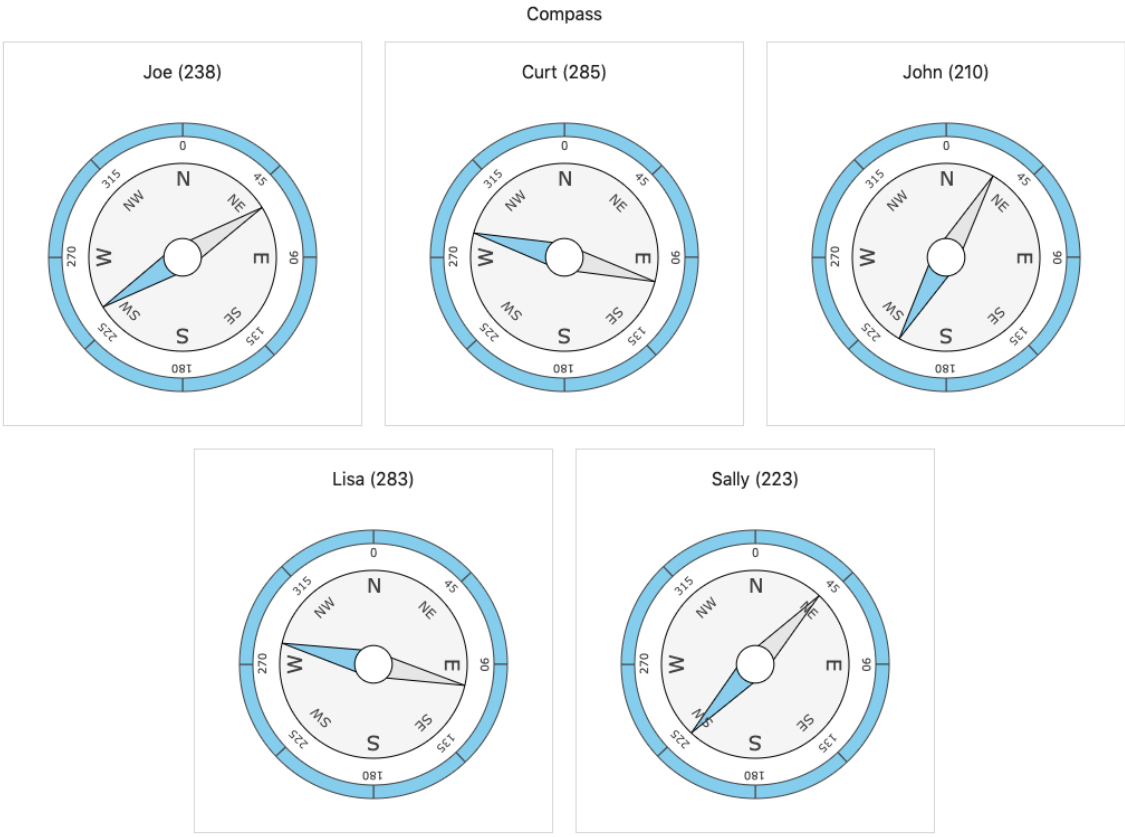
Table 30 *Parameters for Gauges*

Parameter	Description
datasource	Specify an event collection or event stream to feed the gauges.
segments	Specify the number of segments to display in the gauge. The default value is 3.
value	Specify the field value that contains the numeric value to display in the gauge.
size	Specify the pixel size of the gauge. The default value is 300.

Parameter	Description
`shape`	Specify a numeric value between 40 and 89 that determines how much of the circle is occupied by the gauge. The default value is 50.
`range`	Specify the start and end values for the range displayed in the gauge. The default value is `(0,100)`.
`columns`	Specify the number of columns to display. The default value is 3.
`color`	Specify the color to use for the leftmost segment in the gauge. The remaining segments are colored with a darker gradient of this value.
`colors`	Specify a color palette to display in the gauge.
`line_width`	Specify the pixel width of the lines used to draw the gauge.
`title`	Specify the chart title.

Compasses

Use a compass to display a numeric value that represents a navigational heading.



Use code like this to create a compass:

```
compass =
visuals.createCompass(broker,heading="total",size=300,title="Compass",columns=3,

reciprocal_color="#e8e8e8",heading_color="#89cff0",

outer_color="#89cff0",bg_color="#f8f8f8",line_width=1)
```

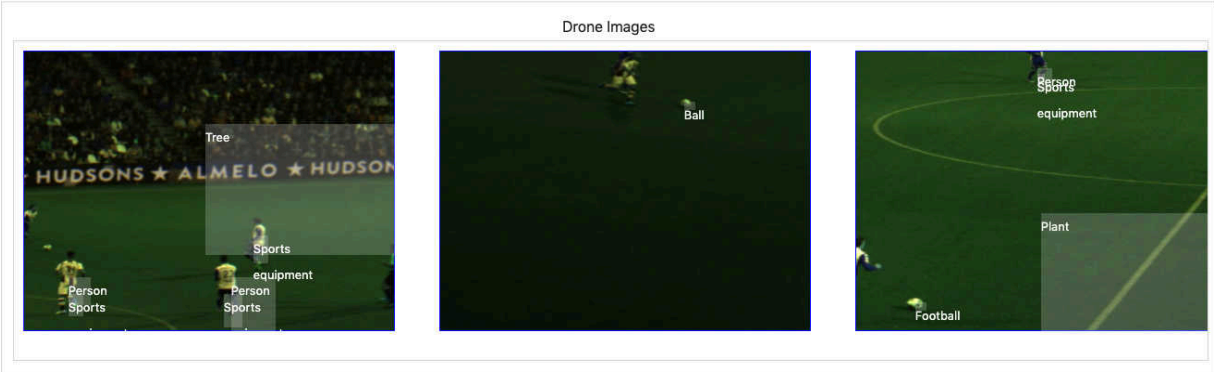
Table 31 *Parameters for Compasses*

Parameter	Description
datasource	Specify the event collection or event stream that feeds the compass.
`heading`	Specify the field value that contains the navigational heading to display in the compass.
`size`	Specify the pixel size of the gauge. The default value is 300.
`columns`	Specify the number of columns to display. The default value is 3.
`heading_color`	Specify the color to use for the heading arrow. The default value is `#89cff0`.

Parameter	Description
`reciprocal_color`	Specify the color to use for the opposite arrow angle. The default value is **`white`**.
`bg_color`	Specify the color to use for the center of the compass. The default value is **`#f8f8f8`**.
`outer_color`	Specify the color to use for the outer circle of the compass. The default value is **`#89cff0`**.
`line_width`	Specify the pixel width of the lines used to draw the gauge. The default value is 1.
`title`	Specify the chart title.

Images

You can use the Images component to display event image data.



When object detection data exists in the event, the Images component displays annotations inside the image that denote the contained objects.

Use code like this to display images.

```
images = visuals.createImages(stream,image="_image_",
                                        image_width=400,
                                        image_height=400,
                                        orientation="horizontal",
                                        image_border="2px solid blue",
                                        label_color="white")
```

Table 32 *Parameters for Images*

Parameter	Description
`datasource`	Specify the event collection or event stream that feeds the image data.

Parameter	Description
`image`	Specify the field value that contains the image data to display in the component.
`image_width`	Specify the display width of each image. The default value is 300.
`image_height`	Specify the display height of each image. The default value is 300.
`orientation`	Specify the orientation of the images, **`vertical`** (default) or **`horizontal`**.
`image_border`	Specify the border around each image. Use standard CSS syntax (for example, **`border="1px solid blue"`**.
`label_color`	Specify the color to use for the labels displayed when object detection data is present in the image.
`title`	Specify the chart title.

Tables

You can display as many event fields as you choose within a table. If you do not specify a set of `values`, all event fields are displayed.

Broker Alerts

brokerName	frontRunningBuy	frontRunningSell	openMarking	closeMarking	restrictedTrades	total
Joe	45	73	85	0	35	238
Curt	52	58	105	10	60	285
John	20	50	90	15	35	210
Lisa	50	63	115	15	40	283
Sally	25	53	75	15	55	223

Use code like the following to create tables.

```
table =
visuals.createTable(brokerAlerts,values=["brokerName","total","restrictedTrades"],
                                    title="Broker Alerts")
```

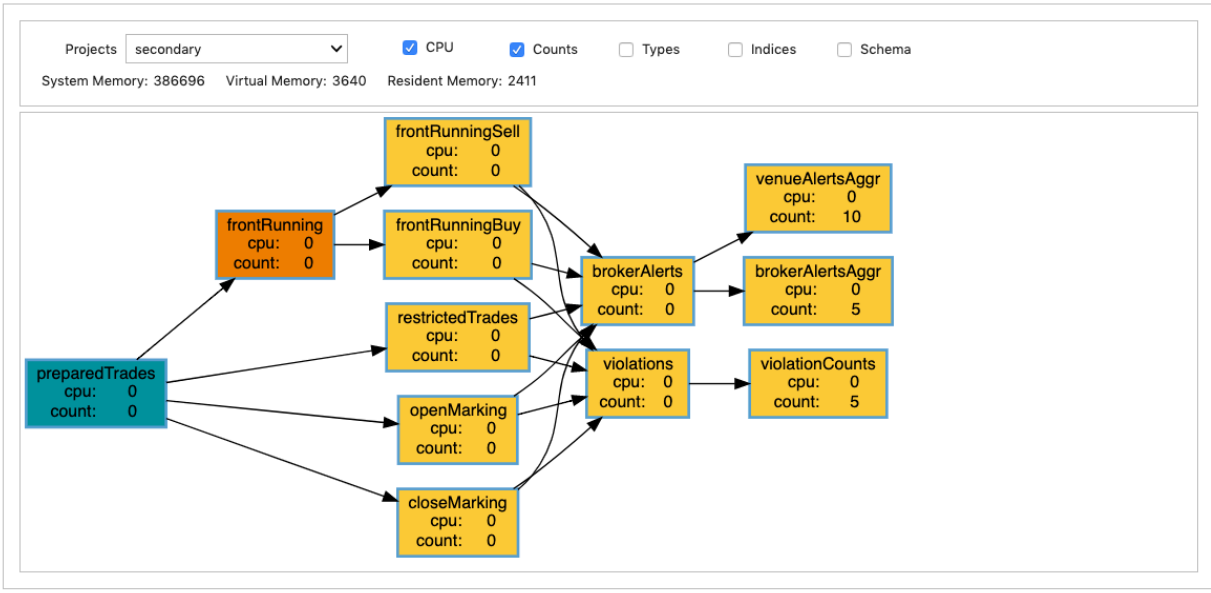
Table 33 *Parameters for Tables*

Parameter	Description
`datasource`	Specify the event collection or event stream that feeds the table.
`values`	Specify the event fields to be displayed in the table.

Parameter	Description
`title`	Specifies the chart title.

Event Stream Processing Model Viewer

The Model Viewer displays a graphical representation of the model in the form of a directed graph. You can view as many attributes of each window as you choose. Some model attributes are dynamic and some are static. The Model Viewer also displays ESP server memory usage information. System memory, virtual memory, and resident memory information is displayed above the directed graph. This information changes as the model processes the event stream.



Use code like the following to create a model viewer:

```
viewer = visuals.createModelViewer(conn,cpu=True,counts=True,

type=False,index=False,cpu_color="lightblue")
```

Table 34 *Parameters for the Model Viewer*

Parameter	Description
`cpu`	Display the CPU usage of the window.
`counts`	Display the number of events currently in the window.
`type`	Display the window type.
`index`	Display the window index type.

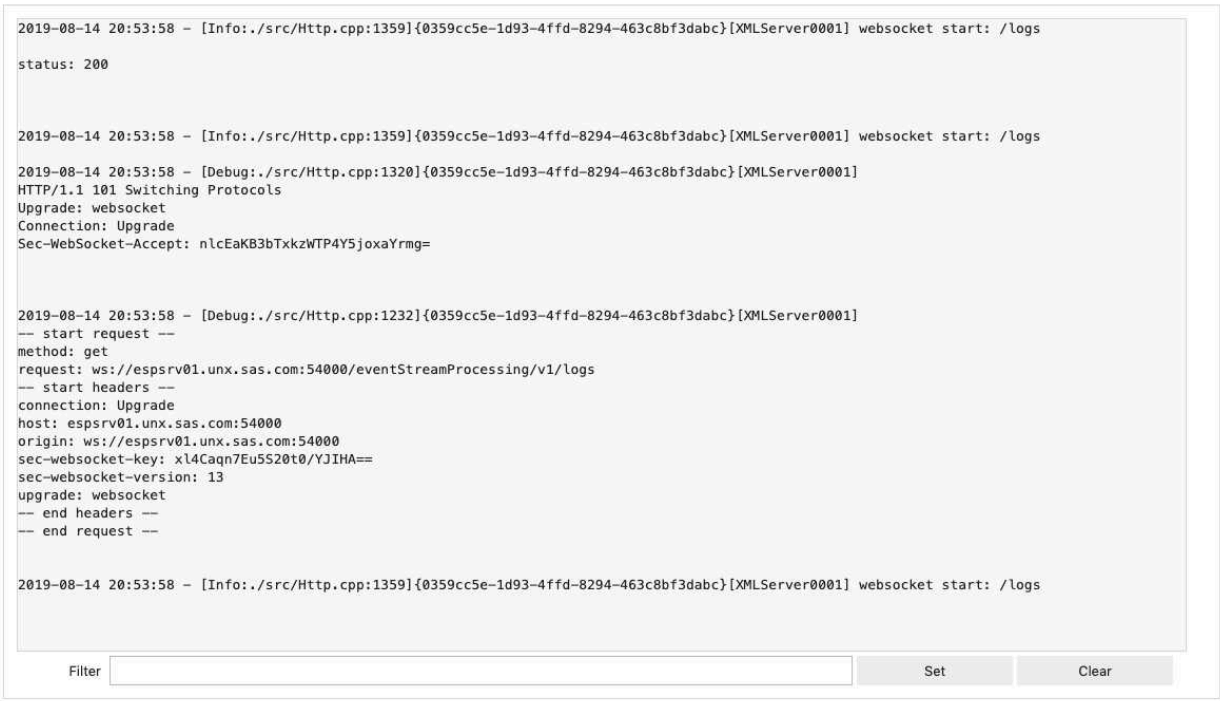
Parameter	Description
schema	Display the window index type.
cpu_color	Use a color gradient that begins with the specified value to color models nodes by CPU usage.

By default, the model viewer displays all projects running in the ESP server. However, if you want to focus on a specific project, then specify that before displaying the model viewer:

```
viewer =
visuals.createModelViewer(conn,cpu=True,counts=True,type=False,index=False,cpuColor=Tr
ue)
viewer.project = "primary"
viewer
```

Event Stream Processing Log Viewer

Use the Log Viewer object to view a live ESP server log. Messages appear at the same time that they appear on the console. The Log Viewer displays the most recent log messages at the top.



Use code like this to create a log viewer:

```
visuals.createLogViewer(conn,height="600px",filter="")
```

Table 35 Parameters for a Log Viewer

Parameter	Description
conn	Specify the connection object.
`height`	Specify the height of the log viewer (for example, **`"500px"`**). The default value is "200px".
`filter`	Specify a filter to use on log entries. When set, you get a text field in which you can change the filter.

Laying Out Visualizations

You can use the `HBox` and `VBox` widgets of `ipywidget` to layout your visualizations.

The `HBox` is a horizontal box. When you add visualizations to an `HBox`, they appear in the cell from left to right.

The `HBox` is a vertical box. When you add visualizations to a `VBox`, they appear in the cell from top to bottom.

For example:

```
import ipywidgets as widgets

brokerChart = visuals.createBubbleChart(brokerAlerts,
                                        y="restrictedTrades",
                                        size="total",
                                        color="frontRunningSell",
                                        title="Broker Alerts")

venueMap = visuals.createMap(venueAlerts,
                             lat="lat",
                             lon="lon",
                             size="count",
                             color="count",
                             title="Venue Alerts",
                             marker_opacity=.6,
                             base_color="#F98E85",
                             marker_border=False,
                             center=(39,-98),
                             zoom=4,
                             popup=["city","count"],
                             sizes=(1,50))

brokerBar = visuals.createBarChart(brokerAlerts,
                                   orientation="horizontal",
                                   y=["total","restrictedTrades"],
                                   title="Broker Alerts")

colors = ["#e03531","#ff684c","#f0bd27","#51b364","#8ace7e"]
```

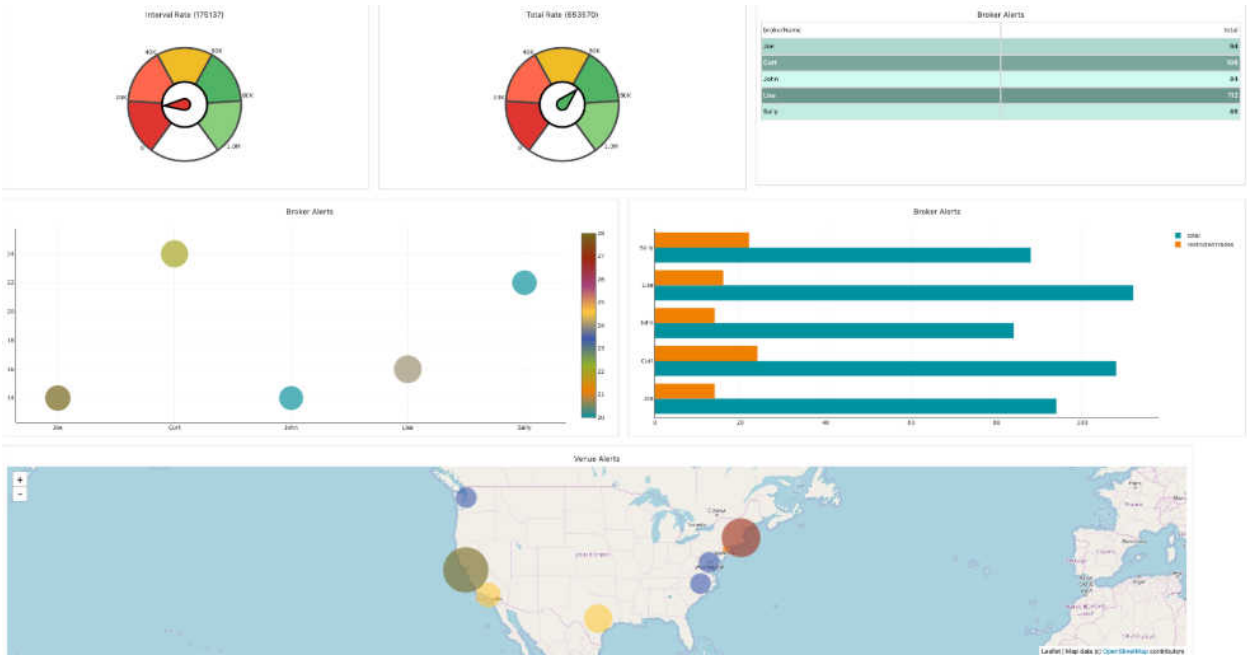
```
intervalRate = visuals.createGauge(rates,
                                   segments=5,
                                   value="intervalRate",
                                   size=300,
                                   shape=80,
                                   colors=colors,
                                   title="Interval Rate",
                                   range=(0,1000000),
                                   width="30%")

totalRate = visuals.createGauge(rates,
                                segments=5,
                                value="totalRate",
                                size=300,
                                shape=80,
                                colors=colors,
                                title="Total Rate",
                                range=(0,1000000),
                                width="30%")

table = visuals.createTable(brokerAlerts,
                            title="Broker Alerts",
                            values=["brokerName","total"],
                            color="total",
                            start_color="#CFFBF3",
                            width="40%")

widgets.VBox([widgets.HBox([intervalRate,totalRate,table]),
              widgets.HBox([brokerChart,brokerBar]),venueMap])
```

That code produces a layout like this:



Scoring an Offline SAS Cloud Analytic Services Model with ESPPy

Overview

You can use SAS Cloud Analytic Services (CAS) to train a model in Python with the SAS Scripting Wrapper for Analytics Transfer (SWAT) package. With ESPPy, you can then score the CAS model during the same Python session.

For more information about SWAT, see the online documentation and the README file.

The following example demonstrates the compatibility of CAS and SAS Event Stream Processing in Python with a deep neural network model trained in CAS and scored in SAS Event Stream Processing. To follow along with the example in this section, open **`Iris_FitStat.ipynb`** in JupyterLab. The sample data set iris.csv can be found at http://support.sas.com/documentation/onlinedoc/viya/examples.htm.

Training and Storing a CAS Model with SWAT

Setting Up SWAT

Install SWAT from https://github.com/sassoftware/python-swat. Like ESPPy, SWAT is open source and supported for Python 2.7 and 3.4+.

For more instructions about setting up and using SWAT, see *Getting Started with SAS Viya for Python*. For more information about SWAT and the full SWAT API reference, see https://sassoftware.github.io/python-swat/.

Connecting to a CAS Server

To use CAS action sets in Python, you need to connect to a running CAS server. For information about starting a CAS server, see *SAS Cloud Analytic Services: Fundamentals*.

Start by importing the `swat` module:

```
In [1]:
import swat as sw
```

Connect to the running CAS server and load the action sets `deepNeural` and `astore` to the session:

```
In [2]:
s = sw.CAS('cas.server', 5570)
s.loadactionset('deepNeural')
s.loadactionset('astore')
```

```
NOTE: Added action set 'deepNeural'.
NOTE: Added action set 'astore'.
```

Out [2]:

§ actionset

astore

elapsed 0.00419s · mem 0.195MB

The `deepNeural` action set includes actions for modeling and scoring with deep neural networks. The `astore` action set enables you to store the scoring models to an analytic store.

You can create a client-side CAS table object using the `CASTable` constructor:

```
In [3]:
iris_table = s.CASTable('iris', replace=True, blocksize=1)
```

The CAS table `iris_table` does not exist on a CAS server. The `read_csv(path,[castout=None])` method of the CAS connection object `s` calls the `pandas.read_csv()` method from Pandas to create a DataFrame of the CSV data. The method also creates a CAS table and stores the DataFrame in the CAS table.

Use the `casout` argument of the `read_csv` method to specify that you want to create a CAS table using the CAS table object:

```
In [4]:
iris_data = s.read_csv('iris.csv', casout=iris_table)
```

```
NOTE: Cloud Analytic Services made the uploaded file available as table IRIS in caslib
CASUSER(username).
NOTE: The table IRIS has been created in caslib CASUSER(username) from binary data uploaded to
Cloud Analytic Services.
```

After you have loaded the data to a table on the server, you can return descriptive information about your data using a number of actions.

The `columnInfo` action, for example, returns information about the columns of the table that you specify:

```
In [5]:
iris_table.columnInfo(table='iris')
```

Out [5]:

§ ColumnInfo

	Column	ID	Type	RawLength	FormattedLength	NFL	NFD
0	Species	1	varchar	10	10	0	0
1	SepalLength	2	double	8	12	0	0
2	SepalWidth	3	double	8	12	0	0
3	PetalLength	4	double	8	12	0	0
4	PetalWidth	5	double	8	12	0	0

elapsed 0.0021s · mem 0.704MB

The `freq` action returns the frequency distribution in the table for species.

```
In [6]:
iris_table[['species']].freq()
```

Out [6]:

§ Frequency

Frequency for IRIS

	Column	CharVar	FmtVar	Level	Frequency
0	Species	Setosa	Setosa	1	50.0
1	Species	Versicolor	Versicolor	2	50.0
2	Species	Virginica	Virginica	3	50.0

elapsed 0.83s · user 30.3s · sys 12.7s · mem 1.62e+04MB

Training a Simple Deep Neural Network Model

You can use the data in `iris_table` to train a deep neural network model.

```
In [7]:
nloOpts = dict(mode=dict(type='SYNCHRONOUS'), algorithm=dict(method='momentum',     #1
                                                             clipgradmin=-1000,
                                                             clipgradmax=1000,
                                                             learningRate=0.01,
                                                             lrpolicy='step',
                                                             stepsize=8,
                                                             useLocking=True),
               dropoutType='inverted', miniBatchSize=1, maxEpochs=10, logLevel=2)

iris_table.dnnTrain(inputs={'SepalLength', 'SepalWidth', 'PetalLength',
'PetalWidth'},    #2
           target='Species', hidden=10, optimizer=nloOpts, nThreads=2,
           modelOut=dict(name='iris_model', replace=True),
```

```
        modelWeights=dict(name='iris_model_weights', replace=True))
```

1 Create a dictionary with entries that specify the optimization settings.

2 Call the `dnnTrain` action on `iris_table`. For a full list of parameters, see *SAS Visual Data Mining and Machine Learning: Deep Learning Programming Guide*.

```
NOTE:  The Synchronous mode is enabled.
NOTE:  The total number of parameters is 83.
NOTE:  The approximate memory cost is 1.00 MB.
NOTE:  Initializing each layer cost       0.00 (s).
NOTE:  The total number of threads on each worker is 2.
NOTE:  The total minibatch size per thread on each worker is 1.
NOTE:  The maximum minibatch size across all workers for the synchronous mode is 2.
NOTE:  Target variable: Species
NOTE:  Number of levels for the target variable:      3
NOTE:  Levels for the target variable:
NOTE:  Level      0: Versicolor
NOTE:  Level      1: Virginica
NOTE:  Level      2: Setosa
NOTE:  Number of input variables:     4
NOTE:  Number of numeric input variables:      4
NOTE:  Epoch           Learning Rate     Loss    Fit Error      Time (s)
NOTE:          0            0.01       0.5339       0.18         0.24
NOTE:          1            0.01       0.3008     0.1067         0.00
NOTE:          2            0.01       0.2228     0.0867         0.00
NOTE:          3            0.01         0.18     0.0667         0.00
NOTE:          4            0.01       0.1521     0.0533         0.00
NOTE:          5            0.01       0.1331     0.0467         0.00
NOTE:          6            0.01       0.1196     0.0333         0.00
NOTE:          7            0.01       0.1098     0.0333         0.00
NOTE:          8           0.001       0.1048     0.0267         0.00
NOTE:          9           0.001       0.0936       0.04         0.00
NOTE:  The optimization reached the maximum number of epochs.
NOTE:  The total time is       0.26 (s).
```

```
Out [7]:
```

§ ModelInfo

	Descr	Value
0	Model Name	iris_model
1	Model Type	Deep Neural Network
2	Number of Layers	3
3	Number of Input Layers	1
4	Number of Output Layers	1
5	Number of Fully Connected Layers	1
6	Number of Weight Parameters	70
7	Number of Bias Parameters	13
8	Total Number of Model Parameters	83
9	Approximate Memory Cost for Training (MB)	1

§ OptIterHistory

	Epoch	LearningRate	Loss	FitError
0	0.0	0.010	0.533939	0.180000
1	1.0	0.010	0.300838	0.106667
2	2.0	0.010	0.222780	0.086667
3	3.0	0.010	0.179952	0.066667
4	4.0	0.010	0.152108	0.053333
5	5.0	0.010	0.133075	0.046667
6	6.0	0.010	0.119613	0.033333
7	7.0	0.010	0.109759	0.033333
8	8.0	0.001	0.104791	0.026667
9	9.0	0.001	0.093573	0.040000

§ OutputCasTables

	casLib	Name	Rows	Columns	casTable
0	CASUSER(username)	iris_model	35	5	CASTable('iris_model', caslib='CASUSER(username...
1	CASUSER(username)	iris_model_weights	83	3	CASTable('iris_model_weights', caslib='CASUSER...

elapsed 1.07s · sys 0.219s · mem 3.69MB

Storing a Model in an Analytic Store

After the model has been trained, you can store the model in an analytic store with the `dnnExportModel` action:

```
In [8]:
s.dnnExportModel(model='iris_model', initWeights='iris_model_weights',
casout=dict(name='iris_astore',

replace=True))
```

```
NOTE: Wrote 8229 bytes to the savestate file iris_astore.
```

```
Out [8]:
```

§ OutputCasTables

	casLib	Name
0	CASUSER(username)	iris_astore

elapsed 2.35s · sys 0.109s · mem 2.37MB

Write the analytic store to an analytic store file on your local disk. Make sure that the ESP server can access the location of the analytic store file.

```
In [9]:
iris_ast = s.astore.download(rstore='iris_astore')
with open('iris_dnn.astore','wb') as file:
    file.write(iris_ast['blob'])
```

Use the `describe` action of the Analytic Store action set to return information about an analytic store.

```
In [10]:
s.astore.describe(rstore='iris_astore')
```

Out [10]:

§ Key

	Key
0	10502B119BF83CC8996E9FEE392EB4E876C084C7

§ Description

	Attribute	Value
0	Analytic Engine	deeplearn
1	Time Created	30Apr2018:13:18:58

§ InputVariables

Input Variables

	Name	Length	Role	Type	RawType	FormatName
0	PetalLength	8.0	Input	Interval	Num	
1	SepalLength	8.0	Input	Interval	Num	
2	SepalWidth	8.0	Input	Interval	Num	
3	PetalWidth	8.0	Input	Interval	Num	

§ OutputVariables

Output Variables

	Name	Length	Type	Label
0	P_SpeciesVersicolor	8.0	Num	Predicted: Species=Versicolor
1	P_SpeciesVirginica	8.0	Num	Predicted: Species=Virginica
2	P_SpeciesSetosa	8.0	Num	Predicted: Species=Setosa
3	I_Species	10.0	Char	Into: Species

elapsed 2.12s · sys 0.125s · mem 2.87MB

Loading an Offline Analytic Store Model in Python

Connect to an ESP Server

With ESPPy and Graphviz installed, import the modules that you need to use ESPPy. This example uses Pandas to read in data and `matplotlib` to visualize the data.

```
In [11]:
import esppy
%matplotlib inline
import pandas as pd
```

Use Pandas to read in the sample data:

```
In [12]:
iris_data = pd.read_csv('iris.csv', header=0)
```

Connect to a running ESP server with the `ESP()` constructor:

```
In [13]:
esp = esppy.ESP('http://espserver:54321')
esp.server_info()
```

```
Out [13]:
{'analytics-license': True,
 'engine': 'esp',
 'http': 54321,
 'pubsub': 64321,
 'version': '5.2'}
```

The ESP server should be empty if it has just been started.

```
In [14]:
esp.get_projects()
```

```
Out [14]:
{}
```

Creating a Model

SAS Event Stream Processing uses streaming analytics windows to score offline models and calculate analytical and descriptive statistics. For more information about streaming analytics in SAS Event Stream Processing, see *SAS Event Stream Processing: Using Streaming Analytics*.

Create a project with ESPPy:

```
In [15]:
fsProject = esp.create_project('fsProject')
```

Use the Window methods to create the windows that make up the project.

First, create a Source window to read in the iris data. The schema of the Source window must correspond to the columns of the `iris` CAS table.

```
In [16]:
src = esp.SourceWindow(schema=('id*:int64', 'Species:string','SepalLength:double',
'SepalWidth:double',
                               'PetalLength:double', 'PetalWidth:double'),
                       index_type='empty', insert_only=True)

fsProject.windows['w_data'] = src
```

Create a Model Reader window to read the analytic store model to the project.

```
In [17]:
model_reader = esp.ModelReaderWindow()
fsProject.windows['w_reader'] = model_reader
```

Create a Source window to read in the model data.

```
In [18]:
model_request = esp.SourceWindow(schema=('req_id*:int64', 'req_key:string',
'req_val:string'),
                                 index_type='empty', insert_only=True)
fsProject.windows['w_request'] = model_request
```

Create a Score window that uses the offline analytic store model to score the iris data.

```
In [19]:
model_score = esp.ScoreWindow(schema=('id*:int64', 'Species:string', 'I_Species:string',
                                      'P_SpeciesVERSICOLOR:double',
'P_SpeciesVIRGINICA:double',
                                      'P_SpeciesSETOSA:double'))

model_score.add_offline_model(model_type='astore')
fsProject.windows['w_score'] = model_score
```

Note: This schema must match the results from astore.describe.

Create a Calculate window to compute fit statistics for the scored results.

Note: To create online Calculate, Train, and Score windows, use algorithm-specific window constructors of the form *server*.*window_type*.*algorithm*. For a list of the supported algorithms for each streaming analytics window type, access the *server*.*window_type*.supported_algorithms attribute.

```
In [20]:
fitstat = esp.calculate.FitStat(schema=('id*:int64', 'Species:string',
'I_Species:string',
                                'P_SpeciesVersicolor:double',
'P_SpeciesVirginica:double',
                                'P_SpeciesSetosa:double',
'mceOut:double','mcllOut:double'),
classLabels='Versicolor,Virginica,Setosa', windowLength=10)
fitstat.set_inputs(inputs=('P_SpeciesVersicolor:double','P_SpeciesVirginica:double',
                           'P_SpeciesSetosa:double'),
response='Species:string')
fitstat.set_outputs(mceOut='mceOut', mcllOut='mcllOut')
fsProject.windows['w_fitstat'] = fitstat
```

Create another Calculate window that computes the receiver operating characteristic (ROC) curve using the scored results for the classification model.

```
In [21]:
roc = esp.calculate.ROC(schema=('id*:int64', 'Species:string', 'I_Species:string',
                                'P_SpeciesVERSICOLOR:double',
'P_SpeciesVIRGINICA:double',
                                'P_SpeciesSETOSA:double', 'binIdOut*:int64',
'cOut:double'),
                        event='Versicolor', windowLength=10)

roc.set_inputs(input=('P_SpeciesVERSICOLOR:double'),response='Species:string')
roc.set_outputs(binIdOut='binIdOut*:int64',cOut='cOut')

fsProject.windows['w_roc'] = roc
```

Create an Aggregate window that computes the average `mcllOut` by species.

```
In [22]:
agg = esp.AggregateWindow(schema=('Species*:string','AveragemcllOut:double'),pubsub=True)
agg.field_expressions.append('ESP_aAve(mcllOut)')
fsProject.windows['w_agg'] = agg
```

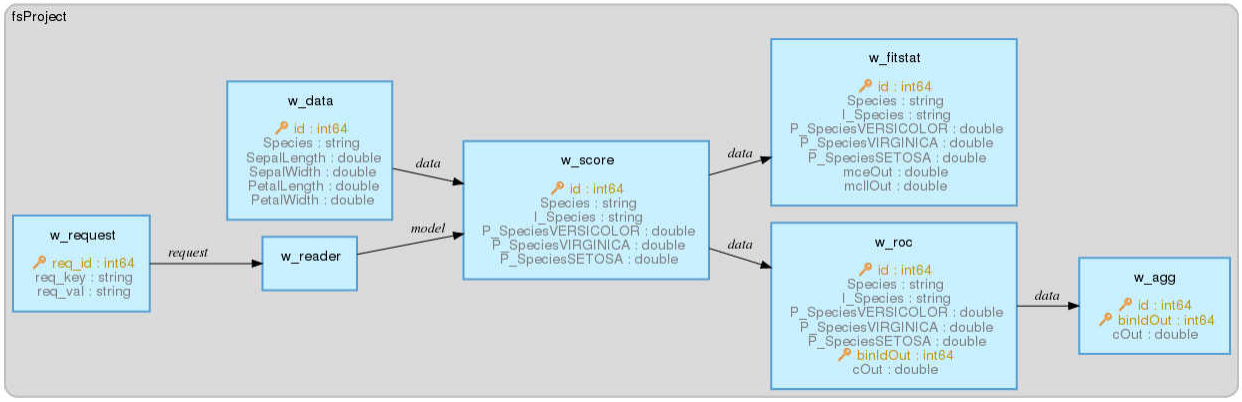
After the windows have been created and added to the project, you can connect them by creating edges.

```
In [23]:
src.add_target(model_score, role='data')
model_request.add_target(model_reader, role='request')
model_reader.add_target(model_score, role='model')
model_score.add_target(fitstat, role='data')
model_score.add_target(roc, role='data')
fitstat.add_target(agg,role='data')
```

Visualize the model in your notebook with the `to_graph()` method:

```
In [24]:
fsProject.to_graph(schema=True)
```

```
Out [24]:
```



Load the model to the ESP server:

```
In [25}:
esp.load_project(fsProject)
esp.get_projects()
```

```
Out [25]:
{'fsProject': Project(name='fsProject')}
```

Publish and Visualize Data Streaming through the ESP Server

Assign a variable to your raw data variable.

```
In [26]:
myiris= iris_data
myiris.index.name='id'
myiris
```

```
Out [26]:
```

	Species	SepalLength	SepalWidth	PetalLength	PetalWidth
id					
0	Setosa	50	33	14	2
1	Setosa	46	34	14	3
2	Setosa	46	36	10	2
3	Setosa	51	33	17	5
4	Setosa	55	35	13	2
5	Setosa	48	31	16	2
6	Setosa	52	34	14	2
7	Setosa	49	36	14	1

Use the `publish_events()` method to publish data to the data events Source window of the model. Use the `subscribe()` method to create a DataFrame for the Source window as the SourceWindow's `data` attribute. After you create the DataFrame in the `data` attribute of a window object, you can use Pandas DataFrame methods on the window object as if it were a DataFrame:

```
In [27]:
src.publish_events(myiris, pause=1000)
src.subscribe()
src.info()
```

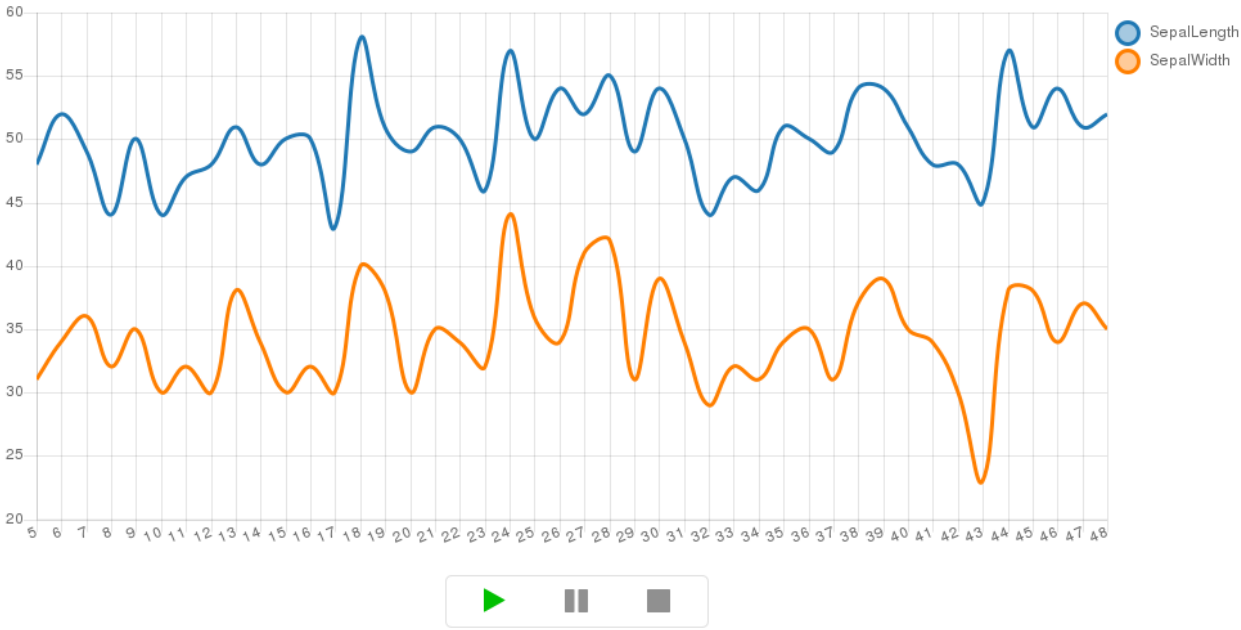
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 0 entries
Data columns (total 5 columns):
Species        0 non-null object
SepalLength    0 non-null float64
SepalWidth     0 non-null float64
PetalLength    0 non-null float64
PetalWidth     0 non-null float64
dtypes: float64(4), object(1)
memory usage: 0.0+ bytes
```

Use the ESPPy streaming charts methods to visualize the data streaming through the model:

```
In [28]:
```

```
src.streaming_line('id', ['SepalLength', 'SepalWidth'], steps=100000.0, interval=1000,
max_data=100)
```

Out [28]:



Publish Offline Model and Model Request Events

After the event stream processing model has been set up, create a publisher to the Source window that processes model and request events. Use the `send` methods of the `Publisher` class to publish the analytic store file created in CAS to the Source window.

```
In [29]:
pub = model_request.create_publisher(blocksize=1, rate=0, pause=0,
                                     dateformat='%Y%m%dT%H:%M:%S.%f', opcode='insert',
format='csv')
pub.send('i,n,1,"action","load"\n')

pub.send('i,n,2,"type","astore"\n')

pub.send('i,n,3,"reference","full_path/iris_dnn.astore"\n')

pub.send('i,n,4,,\n')

pub.close()
```

You can visualize the calculation results from the FitStat and ROC Calculate windows with streaming charts:

```
In [30]:
fitstat.subscribe(mode='streaming')
fitstat.streaming_line(x='id', y=['mcllOut'], steps=100000.0, interval=1000, max_data=50)
```

```
In [31]:
from esppy.plotting import StreamingChart
streamChart=agg.streaming_bar('Species','AveragemcllOut')
```

streamChart