



SAS[®] Event Stream Processing: Python インターフェイスの使用

2020.1 - 2021.1.6

このドキュメントは、ソフトウェアの追加バージョンに適用される場合があります。このドキュメントを [SAS Help Center](#) で開き、バナーのバージョンをクリックすると、使用できるすべてのバージョンが表示されます。

入門ガイド	2
概要.....	2
要件.....	2
ESPPy の設定.....	3
ESP サーバーへの接続	3
ESPPy モジュールのインポート.....	4
ESP サーバーへの接続.....	4
ESP サーバーへの接続後.....	5
Python へのプロジェクトのロード	6
Python でのプロジェクトの作成	9
ESP、Project、ContinuousQuery、および Window クラスのメソッドの使用.....	9
テンプレートの作成と使用.....	13
Python でのプロジェクトの操作	17
概要.....	17
モデルの可視化.....	17
ウィンドウへのデータのパブリッシュ.....	18
イベントジェネレータの使用.....	20
ウィンドウの Pandas DataFrame の操作.....	22
イベントジェネレータの停止.....	24
ESPPy の可視化プログラミングオブジェクト	25
概要.....	25
入門ガイド.....	25

プログラミングオブジェクト	26
ビジュアルインスタンスの作成	37
グラフの作成	39
Event Stream Processing Model Viewer	52
Event Stream Processing Log Viewer	53
可視化のレイアウト	54

入門ガイド

概要

ESPPy は、Python のオープンソースパッケージであり、次の Web サイトで入手できます <https://github.com/sassoftware/python-espp>。ESPPy を使用すると、SAS Event Stream Processing プログラムによってモデルを作成でき、ESP サーバーに接続し、Python オブジェクトとしてプロジェクトとそのコンポーネントと対話できます。これらのオブジェクトには、連続クエリ、ウィンドウ、イベント、ロガー、SAS Micro Analytic Service モジュール、ルーター、分析アルゴリズムが含まれます。ESPPy は ESP プロジェクトとストリーミングチャートのダイアグラムの可視化をサポートする JupyterLab と完全に統合されています。

このドキュメントでは、ESPPy の使用方法について説明します。2 つのコーディング例を順を追って説明します。彼らの Jupyter ノートブックはパッケージで利用できます。例の Python コードは、Anaconda ディストリビューションにてパッケージ化されている Python 3.6 と JupyterLab を使用して開発されました。

[株式取引を処理する例](#)を使用して ESPPy の機能を例で説明します。SAS Event Stream Processing Studio 例の [GitHub リポジトリ](#)から、XML コードおよび関連する CSV 入力ファイルをダウンロードできます。これらのファイルは、このドキュメントで説明する ESPPy プロジェクトの例に使用されます。

ESPPy の詳細については、[README](#) ファイルも参照してください。必要に応じて、[ESPPy の API リファレンス](#)を参照してください。

要件

ESPPy を使用するには、Python 3.4+がシステムにインストールされている必要があります。ESPPy は SAS Event Stream Processing 5.2 以降でのみ動作します。

JupyterLab をインストールするには [JupyterLab のインストール](#)を参照してください。

または、Continuum Analytics の Anaconda プラットフォームを使用することもできます。Anaconda は、Python を含み、JupyterLab へのアクセスや、他のパッケージや環境の広範なライブラリを提供します。Anaconda ディストリビューションは、[ダウンロードサイト](#)から入手できます。

ESPPy で SAS Micro Analytic Service モジュールを使用するには、ESP サーバーを実行するコンピューターシステムで 2 つの環境変数を設定する必要があります。

- MAS_M2PATH は必要な Python プログラム、mas2py.py、へのフルパス名です。
- MAS_PYPATH は、目的の Python 実行可能ファイルへのフルパス名です。

たとえば、

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

SAS Micro Analytic Service の詳細については、[SAS Micro Analytic Service: Programming and Administration Guide](#) を参照してください。

ESPPy の設定

ESPPy をインストールするには、プロジェクトの [README ファイル](#) の指示に従ってください。

ESPPy は、Graphviz を使用して JupyterLab でプロジェクトをチャートとして可視化します。graphviz プログラムを <https://www.graphviz.org/download/> からダウンロードし、システムにインストールします。Graphviz はプロジェクトの可視化に必要であっても、ESPPy は Graphviz を実行する必要はありません。

ESPPy は次のパッケージも使用します。

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

ESP サーバーへの接続

Python で SAS Event Stream Processing プロジェクトをロードして作成するには、実行中の ESP サーバーに接続する必要があります。ESP サーバーの起動と使用の詳細については、“[エッジ環境での ESP サーバーの構成](#)” ([SAS Event Stream Processing: Edge 環境での ESP サーバーの使用](#)) を参照してください。

ESP サーバーを Kubernetes で実行するかどうかによって、サーバーへの接続方法が異なります。次のセクションでは、スタンドアロンまたは Kubernetes ESP サーバーのいずれかで接続する方法について説明します。

ESPPy モジュールのインポート

JupyterLab で ESPPy を動作させるには、esppy モジュールをインポートします。

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

ESP サーバーへの接続

ESP サーバーを起動し、必要なモジュールをインポートした後に、実行中の ESP サーバーに ESP コンストラクタを使用して接続できます。コンストラクタの一般的な構文は、接続先が [スタンドアロン](#) なのか [Kubernetes](#) ESP サーバーなのかによって違います。

スタンドアロンの ESP サーバーへの接続

ESP コンストラクタを使用して、*host* 名と HTTP ポート番号の *port* で実行中のサーバーに接続する ESP サーバーオブジェクトを Python で作成します。サーバーが認証を使用している場合、*username* と *password* を指定します。

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

ESP サーバーの名前が **espserver** で、HTTP ポートが **54321** であるとします。server ラインからの出力は次のようになります。

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

Kubernetes の ESP サーバーへの接続

注: 次の手順では、Kubernetes 環境での作業経験が十分にあること、および稼働中の Kubernetes 環境が利用可能で設定されていることを前提としています。Kubernetes 環境での作業の詳細については、[SAS Event Stream Processing: Kubernetes 環境での SAS Event Stream Processing の使用](#)を参照してください。

重要 Kubernetes 環境で作業する場合は、SAS Event Stream Processing が永続ボリュームにアクセスするように設定する必要があります。SAS Model Manager と SAS Model Studio から SAS Micro Analytic Service のパブリッシング先に分析ストアモデルをパブリッシュするには、永続ボリュームが必要です。

詳しくは、[SAS Event Stream Processing: Kubernetes 環境での SAS Event Stream Processing の使用](#)を参照してください。

- 1 Kubernetes で稼働している ESP サーバーとの通信に使用するプロキシを起動するには、kubectl コマンドを使用します。

```
$ kubectl proxy --accept-hosts=".*"--disable-filter --v=7
```

- 2 ESP コンストラクタを使用して、HTTP ポート番号の *port*、*namespace*、および *project name* で実行中のサーバーに接続する ESP サーバーオブジェクトを Python で作成します。サーバーが認証を使用している場合、**username** と **password** を指定します。

```
In [2]: server = esppy.ESP("k8s-proxy://localhost:port/<namespace>/<project name>",username="<user name>",password="<password>",ca_bundle="_noverify_") server
```

- 3 (オプション) ロードするモデルを含むパラメータを指定することもできます。

- `model_file` にはロードするプロジェクトの XML を含むファイルを指定します。
- `model_data` にはモデル XML ファイルを指定します。

ESP サーバーの名前空間が **espserver** で、指定したプロジェクトが **trades** であるとします。server ラインからの出力は次のようになります。

```
Out [2]:
ESP('https://espserver.ingress-nginx.espdev-m1.espkafkabroker.sashq-r.openstack.sas.com/
SASEventStreamProcessingServer/trades')
```

ESP サーバーへの接続後

- 1 サーバー情報を見るには、server オブジェクトの `server_info` 属性を確認します。

```
[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}
```

- 2 ESP クラスの `get_projects()` メソッドを使用して、プロジェクトの *name* に基づく、または *filter* が適用された、実行中の ESP サーバーにあるプロジェクトのリストを返すには、次のようにします。

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

- 3 実行中の ESP サーバー上のウィンドウのリストを *path*、ウィンドウ *type*、または *filter* で返すには、`get_windows()` メソッドを使用します。

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

- 4 プロジェクトをサーバーにロードしていない場合は、`get_projects()` メソッドと `get_windows()` メソッドは空のディレクトリを返します。

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

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

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

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

Python へのプロジェクトのロード

load_project()メソッドを使用して、実行中の ESP サーバーにプロジェクトを Python オブジェクトとしてロードします。一般的な構文は次のとおりです。

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

プロジェクトをロードするには、次のいずれかを指定します。

- XML 文字列
- プロジェクトオブジェクト
- 有効な XML プロジェクト定義を含むローカルファイル名
- 有効な XML プロジェクト定義を指す URL を含む文字列

ここでは、例としてトレードの処理を考察します。現在の作業ディレクトリにある trades.xml ファイルを使用して、XML プロジェクト定義の変数を定義し、load_project()メソッドの入力としてこの変数を使用できます。

In [6]:

```
project_path = '<file path>trades.xml'
project_xml = project_path
```

server は、ESP クラスのインスタンスを参照する変数として定義されています。

In [7]:

```
trades_project = server.load_project(project_xml, name='trades_proj')
```

load_project()メソッドは、変数 project_xml で参照される文字列の XML を解析し、ESP サーバーのプロジェクトとして XML をロードします。このプロジェクトオブジェクトは、Python では変数名 trades_project で参照されます。

プロジェクトオブジェクトが Python で作成され、プロジェクトが ESP サーバーにロードされると、get_projects()メソッドが ESP サーバーにあるプロジェクトのエントリを含むディクショナリを返します。

In [8]:

```
server.get_projects()
```

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

get_projects()メソッドは、ESP サーバーから情報を取得するため、get_projects()で返されるプロジェクトの名前は trades.xml ファイルの project エLEMENT の name 属性に示されたプロジェクト名に一致します。

`get_windows()`メソッドでサーバーにウィンドウをクエリすると、それぞれのウィンドウのエントリを含むディクショナリが返されます。

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')}
```

`get_windows()`は `Project` クラスのメソッドでもあるため、`get_windows()`を `Project` クラスのメソッドとして使用して、ESP サーバーにあるプロジェクトのウィンドウのディレクトリを取得できます。

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')}
```

プロジェクトのコンポーネントは、Python オブジェクト属性として保存されます。プロジェクトオブジェクトの `queries` 属性には、プロジェクトの連続クエリオブジェクトが含まれます。

In [11]:

```
trades_project.queries
```

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

連続クエリの `windows` 属性には、連続クエリのウィンドウオブジェクトが含まれます。

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')}
```

注: get_projects メソッドと get_windows メソッドは、実行中の ESP サーバーにあるプロジェクトまたはウィンドウを参照するオブジェクトを作成します。これらのオブジェクトの属性の変更は、Python セッションのオブジェクトにのみ影響します。これらのオブジェクトへの変更は、load_project メソッドを使用してサーバーにロードすると、サーバーに表示されます。

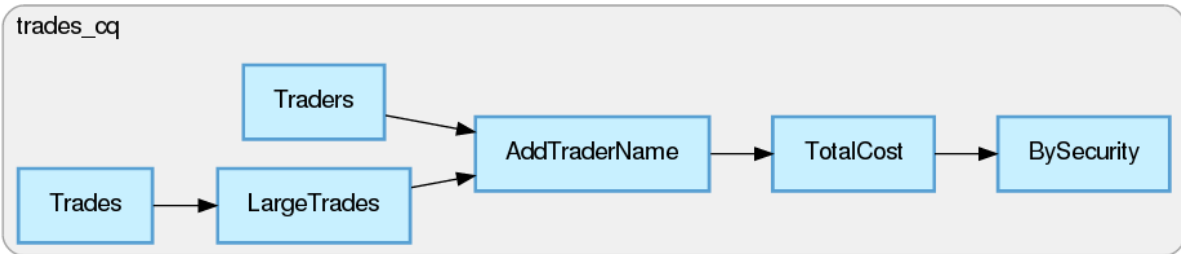
Python での SAS Event Stream Processing プロジェクト作成の詳細については、“[Python でのプロジェクトの作成](#)”を参照してください。

ESPPy をインポートすると自動的にインポートされる Graphviz を使用すると、すべてのプロジェクト、連続クエリ、およびウィンドウオブジェクトのグラフを作成できます。

In [13]:

```
trades_project.queries['trades_cq']
```

```
Out [13]:
```



to_xml()メソッドを使用して、XML フォームでプロジェクトオブジェクトを返せます。pretty=True 属性を使用して、標準形式で XML を表示します。たとえば、次のコードを実行すると、標準 XML 形式の trades_project オブジェクトに対応する XML 定義が返されます。

In [14]:

```
print(trades_project.to_xml(pretty=True))
```

Python でのプロジェクトの作成

ESP、Project、ContinuousQuery、および Window クラスのメソッドの使用

ESP クラス、Project クラス、ContinuousQuery クラス、Window クラスのメソッドを使用して、SAS Event Stream Processing プロジェクトを Python で作成できます。

`create_project()` ESP メソッドを使用してプロジェクトを作成します。`project` に作成するプロジェクト名を指定します。

ESP.create_project(project)

`add_query()` Project メソッドを使用して、連続クエリオブジェクトを作成します。`add_query()` メソッドは、メソッドの呼び出しに使用するプロジェクトに空の連続クエリを追加します。連続クエリの名前を `name` に指定します。

project.add_query(query_name)

各ウィンドウには独自のコンストラクタメソッドがあります。適切な `Window()` メソッドを使用して、ウィンドウオブジェクトを作成します。ウィンドウの `name` に続く個々の値に、ウィンドウタイプ固有のキーワード引数を指定します。

Window(name,<kwargs>)

`add_window()` ContinuousQuery メソッドを使用して、ウィンドウを既存の連続クエリに追加できます。

ここでは、例としてトレードの処理を考察します。`create_project()` メソッドを使用して、プロジェクトを作成することから始めます ESPPy をインポートすると自動的にインポートされる Graphviz は、作成されたオブジェクトのグラフを作成することに注意してください。

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

Out [15]:

my_proj

`add_query()` メソッドを使用して、プロジェクト内に連続クエリを作成します。

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

Out [16]:

trades_cq

SourceWindow コンストラクタを使用して、連続クエリのソースウィンドウを作成します。

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_window()メソッドを使用して、ソースウィンドウを連続クエリに追加します。

In [18]:

```
my_query.add_window(source_trades)
my_project
```

Out [18]:



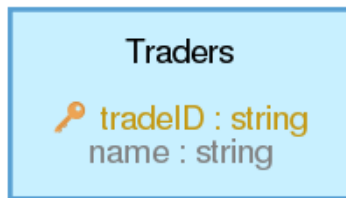
プロジェクト my_proj には、トレードソースウィンドウを含む連続クエリ(trades_cq)が含まれています。

ウィンドウ固有のコンストラクタを使用して、トレードの処理例の残りのウィンドウを作成できます。

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', 'l_security'], ['quantity', 'l_quantity'], ['price', 'l_price'],
                    ['traderID', 'l_traderID'], ['time', 'l_time'], ['name', 'r_name']]
```

```
join.add_field_selection(field_selections[i][0], field_selections[i][1])
```

In [22]:

[illegible]

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

Out [22] :

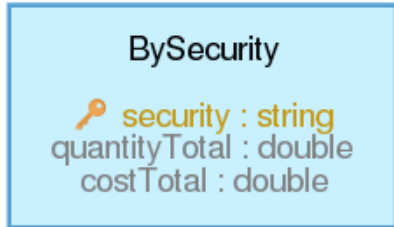


In [23]:

[illegible]

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

```
Out [23]:
```



`add_target()` Window メソッドを使用して、ウィンドウを接続します。`add_target()`メソッドはウィンドウ間にエッジを作成し、エッジの役割を設定します。

In [24]:

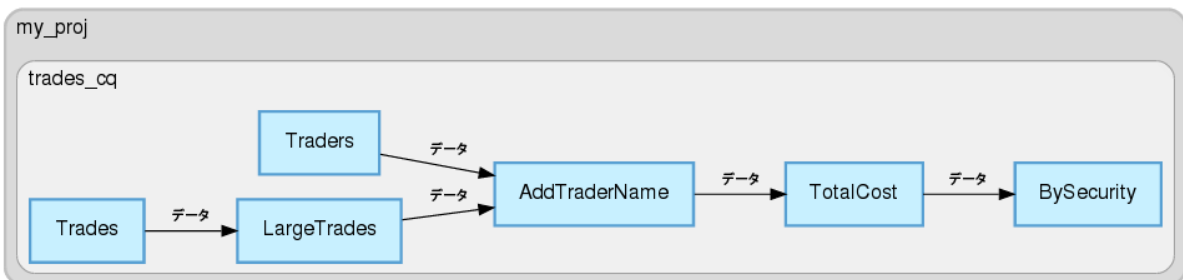
```
source_trades.add_target(filter_large, role='data')
filter_large.add_target(join, role='left')
source_traders.add_target(join, role='right')
join.add_target(compute, role='data')
compute.add_target(aggregate, role='data')
```

`add_window()` ContinuousQuery メソッドを使用して、ウィンドウを連続クエリに追加します。

In [25]:

```
my_query.add_windows(source_trades, source_traders, filter_large, join, compute, aggregate)
my_project
```

```
Out [25]:
```



このプロジェクトはローカルにのみ存在します。これを ESP サーバーにロードする必要があります。

In [26]:

```
server.load_project(my_project)
server.get_projects()
```

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

前と同様に、`to_xml` メソッドに `pretty=True` 引数を使用して、プロジェクトに対応する XML 定義を返すことができます

In [27]:

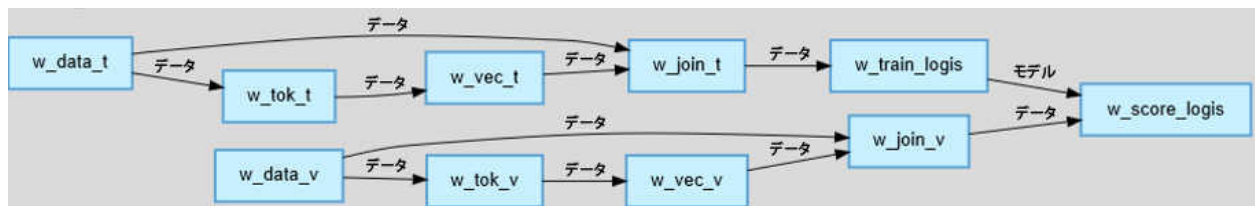
```
print(my_project.to_xml(pretty=True))
```

テンプレートの作成と使用

SAS Event Stream Processing プロジェクトには、特定の 방법으로接続された一連のウィンドウが繰り返し含まれる場合があります。ESPPy を使用して、このようなシーケンスをテンプレートとして使用できます。テンプレートを使用して Python コードを簡略化し、単一の参照でこれらのシーケンスを参照できます。ESPPy に付属の組み込みテンプレートを使用するか、独自のテンプレートを作成できます。

たとえば、商品レビューの感情分析を実行するとします。

図 1 感情分析プロジェクト



このプロジェクトは、入力イベントの 2 つのストリームを実装します。各ストリームのウィンドウは次のとおりです。

- ソースウィンドウ(w_data)は計算ウィンドウ(w_tok)をフィードします。
- その計算ウィンドウは、2 番目の計算ウィンドウ(w_vec)をフィードします。
- 2 番目の計算ウィンドウは結合ウィンドウ(w_join)をフィードします。
- また結合ウィンドウは、ソースウィンドウからイベントを受信します。

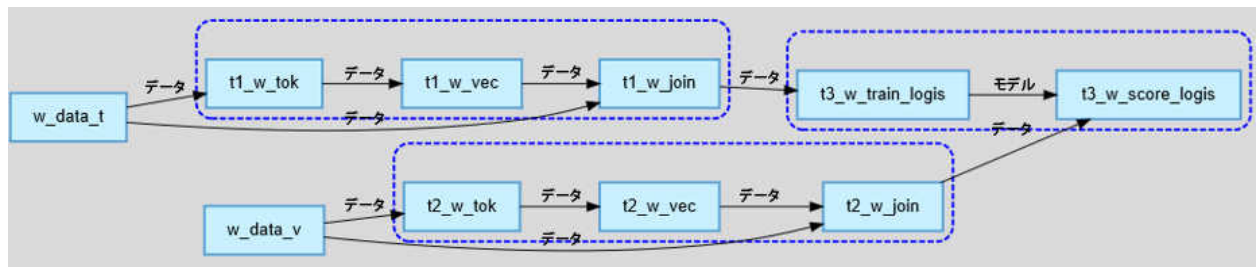
どちらのストリームもスコアウィンドウに収束しますが、最初のストリームはモデルをトレーニングしてから、そのスコアウィンドウにイベントをストリーミングします。

モデル内のウィンドウの 2 つのシーケンスは、ESPPy が提供する組み込みテンプレートで置き換えることができます。

- テキストをトークン化するための計算ウィンドウ、テキストをベクトル化するための計算ウィンドウ、および結合ウィンドウで構成される 1 つのシーケンスが 2 回発生します。このシーケンスは、組み込みの TextEmbedding テンプレートで参照できます。
- 他のシーケンスは、ストリーミングロジスティック回帰のためのトレインウィンドウとスコアウィンドウで構成されます。組み込みのロジスティックテンプレートで参照できます。

次の図は同じプロジェクトを示しており、テンプレートのウィンドウのシーケンスを示しています。

図2 テンプレートが組み込まれた感情分析プロジェクト



ESPPy が提供する組み込みテンプレートの完全なリストは次のとおりです。

表1 組み込みテンプレート

テンプレート	構成ウィンドウ	説明
TextEmbedding	2つの計算ウィンドウ(w_tok、w_vec)、結合ウィンドウ(w_join)	このテンプレートを使用して、ドキュメントテキストを数値ベクトルに変換します(テキストの埋め込みとドキュメントの埋め込み)。テンプレートは、ドキュメントテキストを単語に分解し、単語ごとに単語の埋め込みを取得し、埋め込みを集約してドキュメントの埋め込みを取得します。
Logistic	学習ウィンドウ(train_logis)、スコアウィンドウ(score_logis)	このテンプレートを使用して、ストリーミングロジスティック回帰モデルを学習およびスコアリングします。テンプレートは、ターゲットを持つデータストリームで学習します。次に、別のデータストリームでバイナリ予測を行います。
KeypointsDetection	計算ウィンドウ(w_resized)、モデルリーダーウィンドウ(w_reader)、スコアウィンドウ(w_score)	このテンプレートを使用して、ストリーミング画像のキーポイントを決定します。テンプレートは、各入力画像を指定された幅と高さにサイズ変更します。次に、事前に学習したキーポイント検出モデル(分析ストア形式)を介してそれらをスコアリングします。出力は、画像上のキーポイントの座標(X、Y)です。
ObjectDetection	計算ウィンドウ(w_resized)、モデルリーダーウィンドウ(w_reader)、スコアウィンドウ(w_score)	このテンプレートを使用して、ストリーミング画像内のオブジェクトの境界ボックスを見つけます。テンプレートは、各入力画像を指定された幅と高さにサイズ変更します。次に、事前に学習したオブジェクト検出モデル(分析ストア形式)を介して

テンプレート	構成ウィンドウ	説明
		てそれらをスコアリングします。 出力は、画像内のオブジェクトの境界ボックスの位置(X、Y、W、H)です。

組み込みテンプレートを使用するには、インスタンスを作成して初期化します。

注: 次は、Kubernetes 以外のスタンドアロンサーバーに接続する場合の例です。Kubernetes ESP サーバーへの接続の詳細については、“[Kubernetes の ESP サーバーへの接続](#)”を参照してください。

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

インスタンスを作成した後、テンプレート内で使用可能なウィンドウを確認できます。

```
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)}
```

次のコードは、感情レビュープロジェクト用に、TextEmbedding の事前作成済みテンプレートの 2 つのインスタンスとロジスティックテンプレートの 1 つのインスタンスを作成します。

```
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 事前作成済み TextEmbedding テンプレートのインスタンスを初期化します。
- 2 最初のソースウィンドウ(src)およびテンプレート t1 の間にエッジを追加します。
- 3 t2 という t1 のコピーを作成します。
- 4 2 番目のソースウィンドウ src2 およびテンプレート t2 の間にエッジを追加します。
- 5 事前作成済みロジスティックテンプレートのインスタンスを初期化します。
- 6 テンプレート t1 とテンプレート t3 の間にエッジを追加します。

ESPPy を使用すると、カスタムテンプレートオブジェクトを作成できます。たとえば、次のコードを使用して、感情分析結果に対して後処理を実行するテンプレートを定義できます。

```
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, my_template.output_windows # 7
```

- 1 my_template という名前のカスタムテンプレートのインスタンスを作成します。
- 2 感情の予測値を計算する計算ウィンドウを定義します。
- 3 predicted_y は、予測された P(感情=1)です。
- 4 計算ウィンドウを定義して、それらの計算に適合度統計量を適用します。
- 5 計算ウィンドウと計算ウィンドウの間にエッジを追加します。
- 6 計算ウィンドウと計算ウィンドウをカスタムテンプレートに追加します。
- 7 カスタムテンプレートの input_windows および output_windows を設定します。

カスタムテンプレートは、複数の入力ウィンドウと複数の出力ウィンドウを持つことができます。カスタムテンプレートに接続するウィンドウまたはテンプレートは、input_windows または output_windows で指定する最初の入力ウィンドウに接続します。

たとえば、次のコード行を考えてみましょう。

```
my_template.input_windows=[comp_logis, comp_other]
```

```
my_template.output_windows=[fitstat_logis, fitstat_other]
```

win_instance をこのテンプレートに接続すると仮定します。

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

この場合、win_instance は comp_logis に接続します。

同様に、次のコード行を考えてみましょう。

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

ここで、fitstat_logis は win_instance に接続します。

最後に、カスタムテンプレートの 2 つのインスタンスをセットアップしたとします。次のコード行を考えてみましょう。

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

これにより、最初のテンプレートインスタンスの最初の出力ウィンドウが 2 番目のテンプレートインスタンスの最初の入力ウィンドウに接続されます。

Python でのプロジェクトの操作

概要

XML プロジェクトをロードまたはプロジェクト定義を作成すると、別のメソッドを使用して、SAS Event Stream Processing モデルのデータを可視化、処理、分析できます。

モデルの可視化

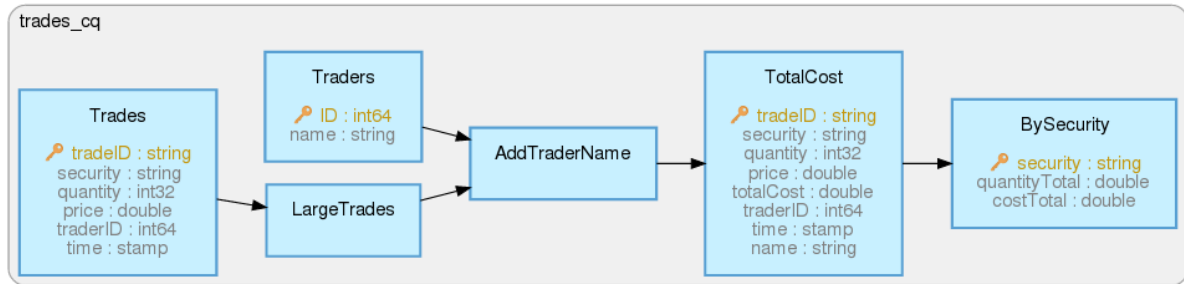
Graphviz により、SAS Event Stream Processing モデルのオブジェクトを可視化できます。

モデルのロード時と作成時に経験したように、単一セルの Jupyter Notebook にあるオブジェクトを要求すると、Graphviz によるグラフィックの可視化が返されます。

また、オブジェクトのグラフをスキーマで表示するために、to_graph()メソッドの引数として schema=true を指定します。

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

```
Out [28]:
```



XML 形式で SAS Event Stream Processing モデルのオブジェクトを返すには、`to_xml()` メソッドを使用します。

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>
```

ウィンドウへのデータのパブリッシュ

プロジェクトオブジェクトのロード後または作成後、SourceWindow サブクラスの `publish_events()` メソッドを使用して、ソースウィンドウにデータを直接、インジェクトできます。

`window.publish_events` (*data* , <引/数>)

トレードの処理例を考えてみましょう。ここでは、2つの異なるソースウィンドウがあり、2つの異なるデータソースからデータを読み込みます。

まず、Python でデータを開きます。[GitHub リポジトリ](#) からダウンロードしたサンプルデータファイル `trades.csv` および `traders.csv` を使用します。現在の作業ディレクトリにあるこれらの CSV ファイルを使用して、データを開いて読み込み、文字列に変数を割り当てます。

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

```

サンプルのトレードデータの列は、トレードソースウィンドウのスキーマに対応しています。

ソースウィンドウに変数を割り当てます。

In [31]:

```

w_trades = trades_project['trades_cq']['Trades']
w_traders = trades_project['trades_cq']['Traders']

```

publish_events() メソッドを呼び出し、データソースとオプションの引数(時間フィールドに dateformat、データ入力の種類に format など)を指定します。

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')

```

get_events() メソッドは Pandas DataFrame を返します。Pandas DataFrame のデータは、ウィンドウに保存されているイベントのスナップショットを示します。

In [33]:

```

w_trades.get_events()

```

```

Out [33]:

```

	security	quantity	price	traderID	time
tradeID					
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

イベントジェネレータの使用

イベントジェネレータを XML 定義から ESPPy にロードすることも、EventGenerator クラスのメソッドを使用して作成することもできます。

イベントジェネレータをロードするには、まず XML 定義を文字列として読み込みます。イベントジェネレータを Python にロードする前に、XML 内の *hostname* と *port* を変更するようにします。また、publish-target エlement で publish-subscribe ポートを必ず指定してください。

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>
```

EventGenerator()コンストラクタを使用して、EventGenerator オブジェクトを作成します。

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')
```

イベントジェネレータを開始するには、プロジェクトセッションを割り当てる必要があります。イベントジェネレータのセッション属性に、イベントジェネレータがイベントをストリームするプロジェクトセッションを定義します。EventGenerator メソッドの save()、initialize()、start()を使用して、ジェネレータの保存、初期化、開始を行います。start()メソッドの rate パラメータは毎秒 10 イベントのみインジェクトするように設定されています。

In [36]:

```
trades_generator.session=trades_project.session
trades_generator.save()
trades_generator.initialize()
trades_generator.start(rate=10)
```

イベントジェネレータを開始した後、トレードソースウィンドウからイベントを取得できます。

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

Out [37]:

	security	quantity	price	traderID	time
tradeID					
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

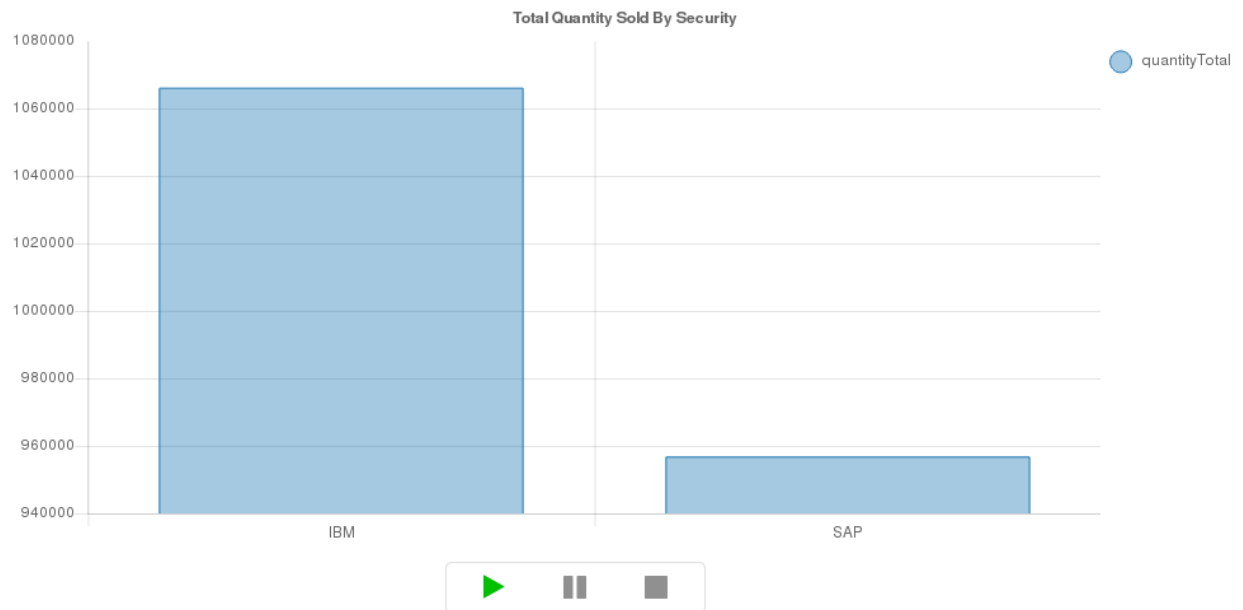
集計ウィンドウ BySecurity を介してストリーミングされるイベントからストリーミング棒グラフを作成するために、StreamingChart をインポートできます。

```
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 ストリーミングチャートを指定すると、関連するウィンドウの独立したサブスクライバーが作成されます。したがって、事前に別のコード行でそのウィンドウをサブスクライブする必要はありません。

Out [38]:



ラインチャート、散布図、円グラフなどのストリーミングチャートについては、<https://sassoftware.github.io/python-espppy/api.html?highlight=streaming%20charts#streaming-charts>を参照してください。

ウィンドウの Pandas DataFrame の操作

Window クラスの `subscribe()` メソッドは、列とインデックスにウィンドウのスキーマが反映された Pandas DataFrame を作成します。この DataFrame は、Window オブジェクトの `data` 属性として保存されます。ウィンドウの DataFrame を作成して保存すると、Window オブジェクトにおいて、このオブジェクトが DataFrame であるかのように、DataFrame 属性へのアクセスと DataFrame メソッドの呼び出しが行えます。

Window メソッドの `subscribe()` を使用してウィンドウのサブスクライバーを作成すると、Window オブジェクトを返す際にウィンドウの `data` 属性のスナップショットが表示されます。

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

```
Out [39]:
```

	security	quantity	price	traderID	time
tradeID					
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

ウィンドウの DataFrame を操作するには、DataFrame クラスのメソッドを使用します。

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]:
```

	security	quantity	price	traderID	time
tradeID					
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

詳細な Pandas API リファレンスについては、<https://pandas.pydata.org/pandas-docs/stable/reference/frame.html> を参照してください。。

イベントジェネレータの停止

ジェネレータを停止したい場合は、stop()メソッドを使用します。イベントジェネレータの停止後、delete()メソッドを使用して、サーバーからジェネレータを削除します。

In [43]:
trades_generator.stop()
trades_generator.delete()

ESPPy の可視化プログラミングオブジェクト

概要

ESPPy は、Jupyter ラボ環境内で動作している ESP モデルを可視化する機能を提供します。次のプログラミングオブジェクトを使用して、さまざまなチャート、ESP モデルビューア、ESP ログビューアを作成できます。

- [ServerConnection](#)
- [EventCollection](#)
- [EventStream](#)
- [Publisher](#)
- [Stats](#)
- [Log](#)

これらのオブジェクトを使用して、次の可視化を構築できます。

- [棒グラフ](#)
- [折れ線グラフ](#)
- [時系列グラフ](#)
- [バブルプロット](#)
- [円グラフ](#)
- [マップ](#)
- [ゲージ](#)
- [コンパス](#)
- [テーブル](#)
- [Event Stream Processing Model Viewer](#)
- [Event Stream Processing Log Viewer](#)

入門ガイド

可視化オブジェクトを使用するには、ESPPy の [README ファイル](#) の指示に従ってください。

プログラミングオブジェクト

ServerConnection オブジェクト

ServerConnection オブジェクトを作成して、可視化をフィードするデータソースを作成します。ServerConnection は、ESP サーバーへの持続的接続です。ESP インスタンスを介してこのオブジェクトのインスタンスを作成します。

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

注: 異なるタイプの環境(Jupyter Lab など)での接続を最適化するために、ESPPy は 2 つの WebSocket エンジンを提供しています。ws4py(デフォルト)と websocket-client です。WebSocket エンジン指定する構文は次の通りです。

```
conn = esp.createServerConnection(websockets="websocket-client")
```

WebSocket エンジン指定しない場合、ESPPy はデフォルトで ws4py を選択します。

可視化 API は、この接続を使用して ESP サーバーと通信します。また、接続はサーバーの状態を監視するために使用されます。接続がアクティブなときにサーバーがダウンすると、API はサーバーが戻るのを待ちます。サーバーが復帰すると、すべてのデータソースが自動的に再接続されます。

ServerConnection メソッド

getEventCollection

EventCollection オブジェクトを作成して返します。

getEventCollection(*path*,*keywordparameters*)

表 2 getEventCollection のパラメーター

パラメーター	説明
<i>path</i>	イベントを取得するウィンドウへのパスを指定します。/project/contquery/window という形式を使用します。

表 3 getEventCollection のキーワードパラメーター

パラメーター	説明
pagesize	コレクションのページサイズを指定します。デフォルト値は 50 です。

パラメーター	説明
filter	コレクションの機能フィルタを指定します。
interval	発生したイベントを配信する前にサーバーが待機する時間をミリ秒単位で指定します。指定しない場合、間隔はデフォルトで 1 秒になります。

例のコード 1 例

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

getEventStream

EventStream オブジェクトを作成して返します。イベントストリームは、イベントストリーム処理ウィンドウの UNIX テールに似ています。

getEventStream(pathkeywordparameters)

表 4 getEventStream のパラメーター

パラメーター	説明
path	イベントを取得するウィンドウへのパスを指定します。 <i>/project/contquery/window</i> という形式を使用します。

表 5 getEventStream のキーワードパラメーター

パラメーター	説明
maxevents	コレクションに保存するイベントの最大数を指定します。デフォルト値は 100 です。
interval	発生したイベントを配信する前にサーバーが待機する時間をミリ秒単位で指定します。指定しない場合、間隔はデフォルトで 1 秒になります。

例のコード 2 例

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

getPublisher

publisher オブジェクトを作成して返します。

getPublisher(path)

表 6 getPublisher の必須パラメーター

パラメーター	説明
<i>path</i>	ソースウィンドウのパスを指定します。 <i>/project/contquery/window</i> という形式を使用します。

例のコード 3 例

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

publishURL

URL から ESP サーバーにデータをパブリッシュします。URL はリクエストでサーバーに送信されます。サーバーは URL からデータをプルし、モデルにインジェクトします。

publishURL(*path, url, keywordparameters*)

表 7 publishURL の必須パラメーター

パラメーター	説明
<i>path</i>	ソースウィンドウのパスを指定します。 <i>/project/contquery/window</i> という形式を使用します。
<i>url</i>	イベントデータを読み取る URL を指定します。イベントデータは、CSV、XML、JSON、またはバイナリ形式で指定できます。

表 8 publishURL のキーワードパラメーター

パラメーター	説明
<i>blocksize</i>	ESP サーバーにイベントをインジェクトするときに使用するイベントブロックサイズを指定します。デフォルト値は 1 です。
<i>wait</i>	true または false を指定して、関数が値を返す前にサーバーでパブリッシュが完了するまで待機するかどうかを示します。デフォルト値は false です。

例のコード 4 例

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

publishDataFrame

ESP サーバーに pandas データフレームをパブリッシュします。

publishDataFrame(*path, dataframe, keywordparameters*)

表 9 publishDataFrame の必須パラメーター

パラメーター	説明
<i>path</i>	ソースウィンドウのパスを指定します。 <i>/project/contquery/window</i> という形式を使用します。
<i>dataframe</i>	パブリッシュするデータフレームを指定します。

表 10 publishDataFrame のキーワードパラメーター

パラメーター	説明
<i>size</i>	各パブリッシュリクエストで送信するイベントの数を指定します。デフォルト値は 100 です。
<i>blocksize</i>	ESP サーバーにイベントをインジェクトするときに使用するイベントブロックサイズを指定します。デフォルト値は 1 です。

例のコード 5 例

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

publishList

Python リストを ESP サーバーにパブリッシュします。

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

表 11 publishList の必須パラメーター

パラメーター	説明
<i>path</i>	ソースウィンドウのパスを指定します。 <i>/project/contquery/window</i> という形式を使用します。
<i>list</i>	パブリッシュする Python リストを指定します。

表 12 publishList のキーワードパラメーター

パラメーター	説明
<i>size</i>	各パブリッシュリクエストで送信するイベントの数を指定します。デフォルト値は 100 です。
<i>blocksize</i>	ESP サーバーにイベントをインジェクトするときに使用するイベントブロックサイズを指定します。デフォルト値は 1 です。

例のコード 6 例

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

getStats

接続の Stats オブジェクトを返します

getStats()

getLog

接続の Log オブジェクトを返します

getLog()

EventCollection オブジェクト

EventCollection は、ステートフルウィンドウのビューです。EventCollection が作成されると、ページサイズが設定されます。このサイズは、サーバーからクライアントに送信されるイベントの最大数です。さらに、サーバーは内部パブリッシュ/サブスクライブインスタンスをセットアップし、ウィンドウのイベントを受信します。イベントが現在のページにない場合、そのイベントは無視されます。それ以外の場合、イベントはクライアントに配信されます。

ServerConnection オブジェクトを介して EventCollection を作成します。

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

イベントコレクションは自身を管理し、変更イベントをそのデリゲートに配信します。コレクション変更通知を受信するには、dataChanged メソッドを実装するデリゲートを登録する必要があります。

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

```
alerts.addDelegate(MyDelegate())
```

dataChanged メソッドは、data パラメーターの新しいデータとともに変更されたコレクションを受信します。

EventCollection メソッド

addDelegate

コレクションの変更通知を受信するデリゲートを追加します。

addDelegate(delegate)

表 13 addDelegate の必須パラメーター

パラメーター	説明
<i>delegate</i>	変更通知を受信するオブジェクトを指定します。

removeDelegate

コレクションからデリゲートを削除します。

removeDelegate(*delegate*)

表 14 removeDelegate の必須パラメーター

パラメーター	説明
<i>delegate</i>	コレクションから削除するオブジェクトを指定します。

getData

イベントのコレクションを返します。データは、イベントキーでソートされたイベントを表すオブジェクトのディクショナリです。

getData()

setFilter

ウィンドウ内のイベントに適用する機能フィルタを設定します。

getData(*filter*)

表 15 setFilter の必須パラメーター

パラメーター	説明
<i>filter</i>	機能フィルタを指定します。

例のコード 7 例

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

load

イベントの現在のページを ESP サーバーからロードします。

load()

first

イベントの最初のページを ESP サーバーからロードします。

first()

last

イベントの最後のページを ESP サーバーからロードします。

first()

next

イベントの次のページを ESP サーバーからロードします。

next()

prev

イベントの前のページを ESP サーバーからロードします。

prev()

EventStream オブジェクト

EventStream は、UNIX テールに似たイベントのフローです。サーバーの ESP モデルでイベントが発生すると、EventStream に配置されます。クライアントはこのストリームを読み取り、発生したイベントを表示できます。ウィンドウが非常に多数のイベントを生成する場合、maxevents および interval を指定することで、ストリームにインジェクトされるイベントの数を調整できます。

maxevents キーワードパラメーターは、一度にストリームに入れられるイベントの数を制限します。interval キーワードパラメーターは、ESP サーバーがイベントをストリームに入れる前に待機する時間をミリ秒単位で指定します。

maxevents を 1000、interval を 1000 ミリ秒(または 1 秒)に設定すると仮定します。ESP サーバーは、パブリッシュ/サブスクライブインターフェイスから 1 秒間イベントを収集します。maxevents を超えるイベントがその間隔で発生する場合、最も古いイベントがドロップされ、ストリームには入れられません。1 秒の終わりに、サーバーは所有しているすべてのイベントをストリームに入れます。処理の負荷が高いクライアント(グラフをレンダリングするクライアントなど)の場合、これによりクライアントは処理する必要のあるイベントの数を制限できます。

削除イベントに関心がない場合は、EventStream を作成するときに ignore_delete=True を指定します。これにより、ストリームによって受信されたすべての削除イベントが破棄されます。

ServerConnection オブジェクトを介して EventStream を作成します。

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

EventStream は変更イベントをそのデリゲートに配信します。ストリーム変更通知が必要な場合、dataChanged メソッドを実装するデリゲートを登録する必要があります。

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

```
rates.addDelegate(MyDelegate())
```

dataChanged メソッドは、data パラメーターの新しいデータとともに変更されたストリームを受信します。

EventStream メソッド

addDelegate

コレクションの変更通知を受信するデリゲートを追加します。

addDelegate(delegate)

表 16 addDelegate の必須パラメーター

パラメーター	説明
<i>delegate</i>	コレクション変更通知を受信するオブジェクトを指定します。

removeDelegate

コレクションからデリゲートを削除します。

removeDelegate(delegate)

表 17 removeDelegate の必須パラメーター

パラメーター	説明
<i>delegate</i>	削除するオブジェクトを指定します。

getData

イベントのコレクションを返します。データは、イベントを表すオブジェクトの配列です。

getData()

setFilter

ウィンドウのイベントで使用する機能フィルタを設定します。

setFilter(filter)

表 18 setFilter の必須パラメーター

パラメーター	説明
<i>filter</i>	機能フィルタを指定します。

例のコード 8 例

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

Publisher オブジェクト

Publisher オブジェクトにより、ユーザーは ESP ソースウィンドウにイベントをパブリッシュできます。1 つ以上の `begin()`、`set()`、`end()` 呼び出しのシーケンスで送信キューにデータを追加できます。または、オブジェクトを送信キューに直接追加する `add()` メソッドを使用できます。Publisher は、`publish()` メソッドが呼び出されるまでパブリッシュされるオブジェクトを保存します。

Publisher メソッド

`begin`

現在のデータを初期化します。

`begin()`

`set`

現在のデータにフィールド値を設定します。

`set(name, value)`

表 19 `set` の必須パラメーター

パラメーター	説明
<i>name</i>	設定するフィールドの名前を指定します。
<i>value</i>	フィールドの値を指定します。

`end`

現在のデータへの入力を閉じます。次に、データをパブリッシュリストに追加します。

`end()`

`add`

データをパブリッシュリストに追加します。

`add(data)`

表 20 追加する必須パラメーター

パラメーター	説明
<i>data</i>	パブリッシュリストに追加するデータオブジェクトを指定します。

publish

パブリッシュリストをソースウィンドウにパブリッシュします。

publish()

Stats オブジェクト

Stats オブジェクトを使用すると、ウィンドウごとにメモリ使用量や CPU 使用率などの ESP サーバー統計情報を監視できます。

データが変更されたときに通知を受け取るには、`handleStats(self,stats)`メソッドを実装するデリゲートオブジェクトをセットアップします。

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

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

Stats 配列には、次の基準の 1 つまたは両方を満たすウィンドウの CPU およびウィンドウイベントカウント情報が含まれています。

- CPU 使用率が最小値を超えています。
- ウィンドウイベントカウントは 0 より大きいです。(インデックス `pi_EMPTY` のウィンドウのイベント数は常に 0 です。)

メモリ情報には、ESP サーバーで現在消費されているシステム、仮想、および常駐の各メモリの値が含まれます。

Stats メソッド

addDelegate

Stats 変更通知を受信するデリゲートを追加します。

addDelegate(delegate)

表 21 必須パラメータ

パラメーター	説明
<i>delegate</i>	変更通知を受信するオブジェクトを指定します。

setOpts

Stats レポートオプションを設定します。

setOpts(keywordparameter)

keywordparameter の値は次のいずれかです。

- **interval** -サーバーがクライアントにデータを送信する間隔(秒単位)。
- **cpu** -報告される最小 CPU 使用率(%)。デフォルト値は 5 です。
- **memory** -メモリ使用率情報を報告するかどうかを判断するのに **true** または **false** を指定します (デフォルトは **true**)。
- **counts** -ウィンドウイベントカウントを報告するかどうかを判断するのに **true** または **false** を指定します(デフォルトは **false**)。
- **config** -ESP サーバー構成情報を報告するかどうかを判断するのに **true** または **false** を指定します (デフォルトは **false**)。

getData

Stats 情報を返します。

getData()

返された Stats データは次のようになります。

```
[
  {
    '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

メモリ情報を返します。

getMemoryData()

返されたメモリデータは次のようになります。

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

Log オブジェクト

Log オブジェクトを使用すると、ESP サーバーログを監視できます。Stats データが変更されたときに通知を受け取るには、`handleLog(log,message)`メソッドを実装するデリゲートオブジェクトを追加する必要があります。

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

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

メッセージパラメーターは、ログメッセージのテキストを含む文字列です。

Log メソッド

addDelegate

Log メッセージを受信するデリゲートを追加します。

addDelegate(delegate)

表 22 addDelegate の必須パラメーター

パラメーター	説明
<i>delegate</i>	Log メッセージを受信するオブジェクトを指定します。

ビジュアルインスタンスの作成

グラフィックを作成するため `esppy` および `Visuals` 両方をインポートする必要があります。

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

`Visuals` オブジェクトは、すべてのチャート、テーブル、およびビューアを作成するために使用されます。`Visuals` インスタンスの作成時に指定できるキーワードパラメーターは次のとおりです。

- `colormap` - カラーテーマ
- `colors` - 文字列形式の色のリスト (["#89cff0", "#0080ff", "#f0bd27", "#ff684c", "#e03531"] など)
- `border` - 各グラフの境界線。標準の CSS 構文を使用してこのプロパティを設定します(`border="1px solid blue"` など)。
- `margin` - 各グラフ内のマージン(ピクセル単位)(デフォルトは"0px")。値は数値の文字列で、後に'px'が続く必要があります(`margin="10px"` など)。

作成時にインスタンスのカラーテーマを指定できます。これらの色は、インスタンスによって作成されたグラフィックのレンダリングに使用されます。

表 23 カラーテーマ

テーマ	エレメント
プロットのカラースケール	■ 黒体 ■ ブルーレッド ■ ブルー ■ アース ■ 電気 ■ グリーン ■ グレイ ■ ホット ■ ジェット ■ ピクニック ■ ポートランド ■ 虹 ■ RdBu ■ レッズ ■ ヴィリディス ■ YlGnBu ■ YlOrRd
Matplotlib カラーマップ	ドキュメント を参照してください。
SAS カラーテーマ	■ sas_base ■ sas_corporate ■ sas_dark ■ sas_highcontrast ■ sas_ignite ■ sas_inspire ■ sas_light ■ sas_marine ■ sas_midnight ■ sas_opal ■ sas_sail ■ sas_snow ■ sas_umstead

テーマ	エレメント
	■ sas_hcb

Visuals インスタンスを作成する呼び出しは、次のようになります。

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

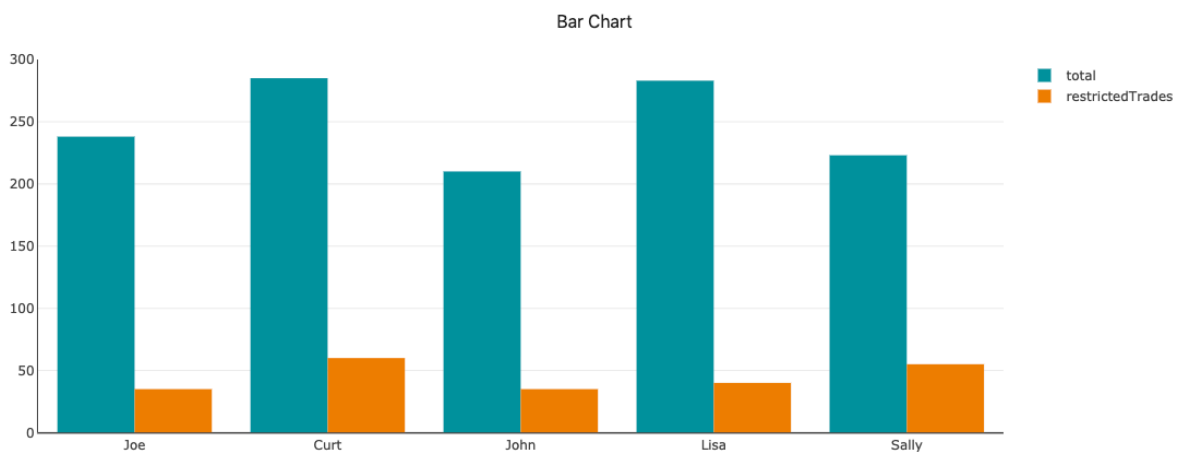
グラフの作成

概要

アクティブなサーバー 接続を有効にして Visuals インスタンスを作成したら、JupyterLab にグラフを含めることができます。適切なチャート作成メソッドを呼び出し、チャートを表示するタイミングと場所にチャートの名前を使用します。

棒グラフ

棒グラフを使用して、X 軸と Y 軸に複数の数値変数をプロットします。



次のようなコードを使用して、棒グラフを作成します。

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

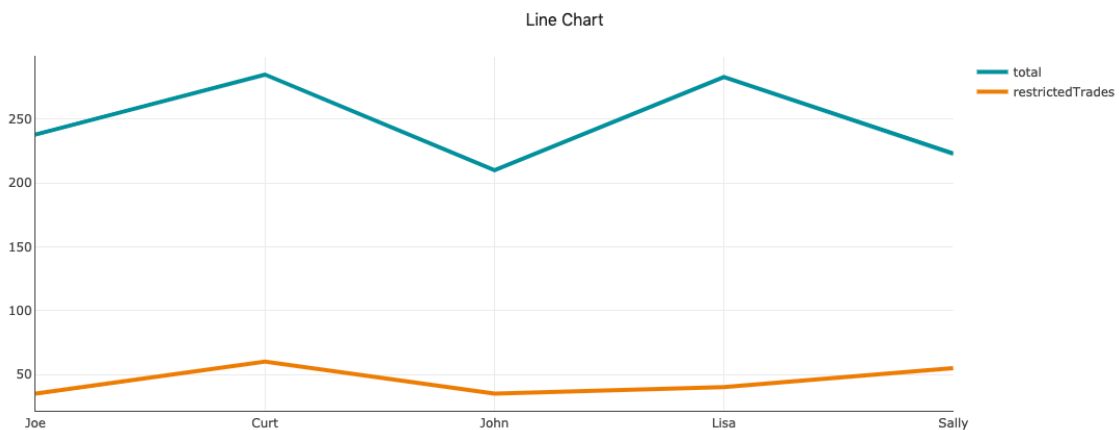
表 24 棒グラフのパラメーター

パラメーター	説明
<i>datasource</i>	棒グラフをフィードするイベントコレクションまたはイベントストリームを指定します。

パラメーター	説明
x	X 軸に表示する分類フィールド値の配列を指定します。x の値を指定しない場合、イベントキー値が使用されます。
y	Y 軸(または orientation の値に応じて X 軸)に表示する数値フィールド値の配列を指定します。
orientation	チャートの向きを指定します。有効な値は vertical (デフォルト)および horizontal です。
title	チャートのタイトルを指定します。

折れ線グラフ

折れ線グラフを使用して、Y 軸に複数の数値変数をプロットします。



次のようなコードを使用して、折れ線グラフを作成します。

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

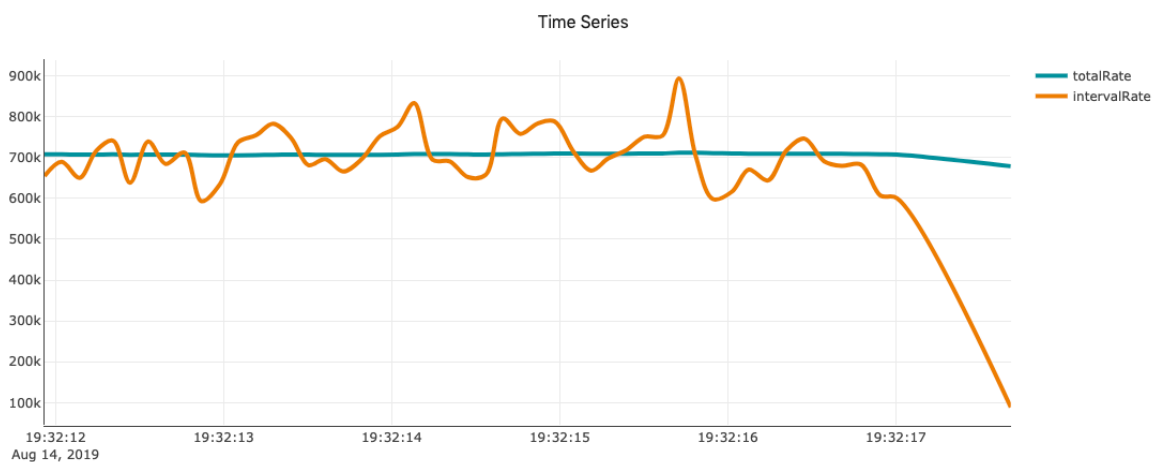
表 25 折れ線グラフのパラメーター

パラメーター	説明
<i>datasource</i>	折れ線グラフをフィードするイベントコレクションまたはイベントストリームを指定します。
x	X 軸に表示する分類フィールド値の配列を指定します。x の値を指定しない場合、イベントキー値が使用されます。
y	Y 軸に表示する数値フィールド値の配列を指定します。

パラメーター	説明
line_width	チャートの線の幅をピクセル単位で指定します。デフォルト値は 2 です。
curved	線を曲線にするかどうかを指定します。有効な値は true または false です(直線です)。
fill	線の下に埋めるかどうかを指定します。有効な値は true または false です。
title	チャートのタイトルを指定します。

時系列グラフ

時系列グラフを指定して、Y 軸に複数の数値変数を、X 軸に日付または時間変数をプロットします。



このグラフを使用するには、date または timestamp いずれかのフィールド値を指定する必要があります。次のようなコードを使用して、時系列グラフを作成します。

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

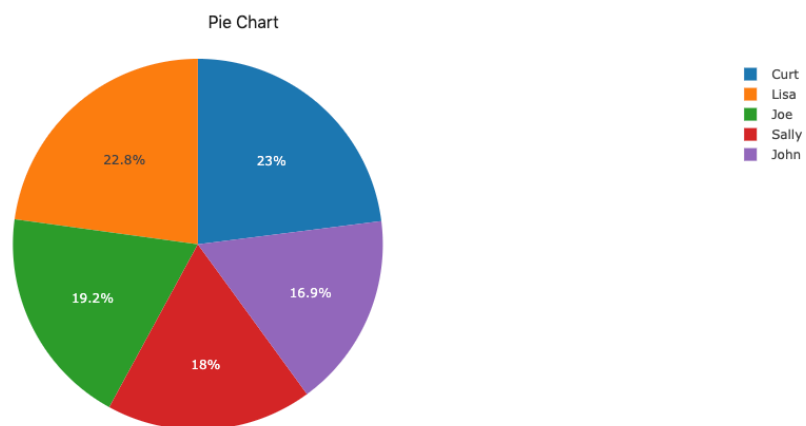
表 26 時系列グラフのパラメーター

パラメーター	説明
datasource	時系列グラフをフィードするイベントコレクションまたはイベントストリームを指定します。
time	タイプ date または timestamp のフィールド値を指定します。
y	Y 軸に表示する数値フィールド値の配列を指定します。

パラメーター	説明
line_width	チャートの線の幅をピクセル単位で指定します。デフォルト値は 2 です。
curved	線を曲線にするかどうかを指定します。有効な値は true または false です(直線です)。
fill	線を下に埋めるかどうかを指定します。有効な値は true または false です。
title	チャートのタイトルを指定します。

円グラフ

円グラフを使用して、数値を円のスライスとして表します。



次のようなコードを使用して、円グラフを作成します。

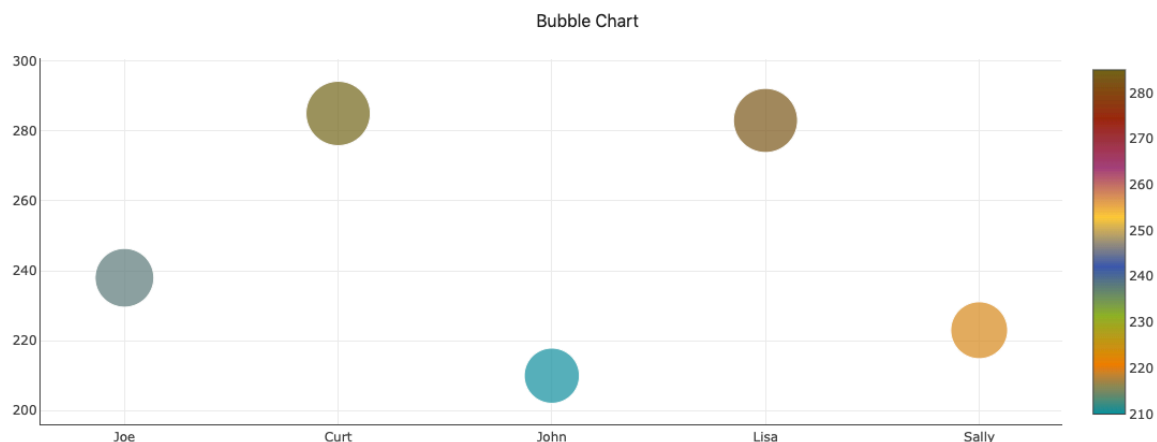
```
brokerChart = visuals.createPieChart(brokerAlerts,value="total",title="Broker Alerts")
```

表 27 円グラフのパラメーター

パラメーター	説明
<i>datasource</i>	円グラフをフィードするイベントコレクションまたはイベントストリームを指定します。
value	円グラフに表示するフィールド値を指定します。
title	チャートのタイトルを指定します。

バブルプロット

バブルプロットを使用して、複数の数値を y、サイズ、色の可視化として表します。



次のようなコードを使用して、バブルプロットを作成します。

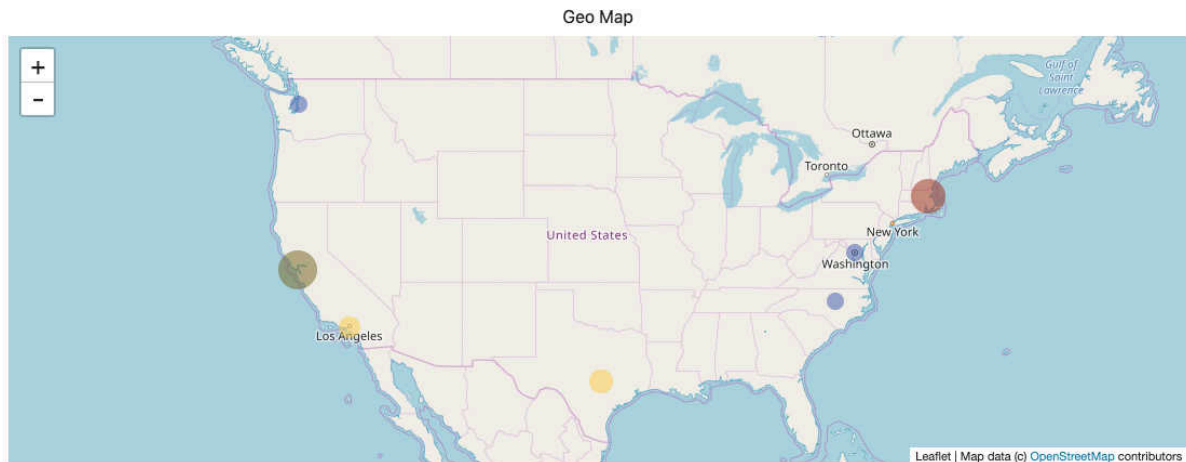
```
brokerChart =
visuals.createBubbleChart(brokerAlerts,y="restrictedTrades",size="total",color="frontRunningSell",title="Broker
Alerts")
```

表 28 バブルプロットのパラメーター

パラメーター	説明
<i>datasource</i>	バブルプロットをフィードするイベントコレクションまたはイベントストリームを指定します。
<i>x</i>	X 軸に表示する分類フィールド値の配列を指定します。x の値を指定しない場合、イベントキー値が使用されます。
<i>y</i>	Y 軸に表示するフィールド値を指定します。
<i>size</i>	各バブルのサイズに使用するフィールド値を指定します。
<i>color</i>	各バブルの色に使用するフィールド値を指定します。
<i>title</i>	チャートのタイトルを指定します。

マップ

マップチャートはJupyter ウィジェット `ipyleaflet` を使用して、イベントデータを表すマーカーでオーバーレイされたマップを表示します。地図上に円や多角形を表示できます。



サークルを追加するには、サークルデータのイベントを含む `EventCollection` オブジェクトを持つ `addCircles` メソッドを呼び出す必要があります。次のキーワードパラメーターを使用します。

- `lat` -円の緯度を表すフィールド
- `lon` -円の経度を表すフィールド
- `radius` -円の半径を表すフィールド
- `text` -ユーザーが円をクリックしたときに表示するテキストを含むフィールド

多角形を追加するには、形状データのイベントを含む `EventCollection` オブジェクトを持つ `addPolygons` メソッドを呼び出す必要があります。次のキーワードパラメーターを使用します。

- `coords` -多角形内の緯度と経度のポイントのスペース区切りリストを含むフィールド
- `text` -ユーザーが多角形をクリックしたときに表示するテキストを含むフィールド
- `order` -ポイントのシーケンス。**`lat_lon`**(デフォルト)の場合、ポイントは緯度、経度の順になります。**`lon_lat`** の場合、ポイントは経度、緯度の順になります。

次のようなコードを使用して、マップチャートを作成します。

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

表 29 マップチャートのパラメーター

パラメーター	説明
<code>datasource</code>	マップチャートにフィードするイベントコレクションまたはイベントストリームを指定します。
<code>lat</code>	緯度を含むフィールド値を指定します。

パラメーター	説明
lon	経度を含むフィールド値を指定します。
size	各マーカーのサイズを取得するために使用するフィールド値、または固定サイズのマーカーの数値を指定します。
color	各マーカーの色を取得するために使用するフィールド値または固定色を指定します。
colormap	マーカーの色付けに使用するカラーテーマを指定します。これか colors を使用しますが、両方は使用しません。
colors	マーカーの色付けに使用する色のリストを指定します (["#89cff0","#0080ff","#f0bd27","#ff684c","#e03531"]など。これか colormap を使用しますが、両方は使用しません。
color_range	色を取得するためにデータ値が比較される開始値と終了値を指定します。これが設定されていない場合、データ値の最小値と最大値が範囲として使用されます。
popup	ユーザーがマーカーをクリックしたときに表示するフィールド値を指定します。
marker_border	マーカーに境界があるかどうか判断するには、 true または false を指定します。
marker_opacity	マーカーの不透明度を決定するには、0～1 の値を指定します。
zoom	マップの初期ズーム値として 1～18 の値を指定します。デフォルト値は 12 です。
center	マップの中心として使用する(lat、lon)のペアを指定します。デフォルト値は (0,0)です。
tracking	初期マーカーが常に中央にくるようにマップを移動するかどうか判断するには、 true または false を指定します。
circle_border_width	円の境界線の幅を数値(ピクセル単位)で指定します。デフォルト値は 1 です。
circle_border_color	円の境界に使用する色を指定します。デフォルト値は black です。
circle_fill_color	円の内側に使用する色を指定します。デフォルト値は white です。
circle_fill_opacity	円の内側の不透明度を数値で指定します。デフォルト値は 2 です。
poly_border_width	多角形の境界線の幅を数値(ピクセル単位)で指定します。デフォルト値は 1 です。

パラメーター	説明
poly_border_color	多角形の境界に使用する色を指定します。デフォルト値は black です。
poly_fill_color	多角形の内側に使用する色を指定します。デフォルト値は white です。
poly_fill_opacity	多角形の内側の不透明度を数値で指定します。デフォルト値は 2 です。

次のコードは、パリの地図を生成し、それにカスタム形状を追加します。

```

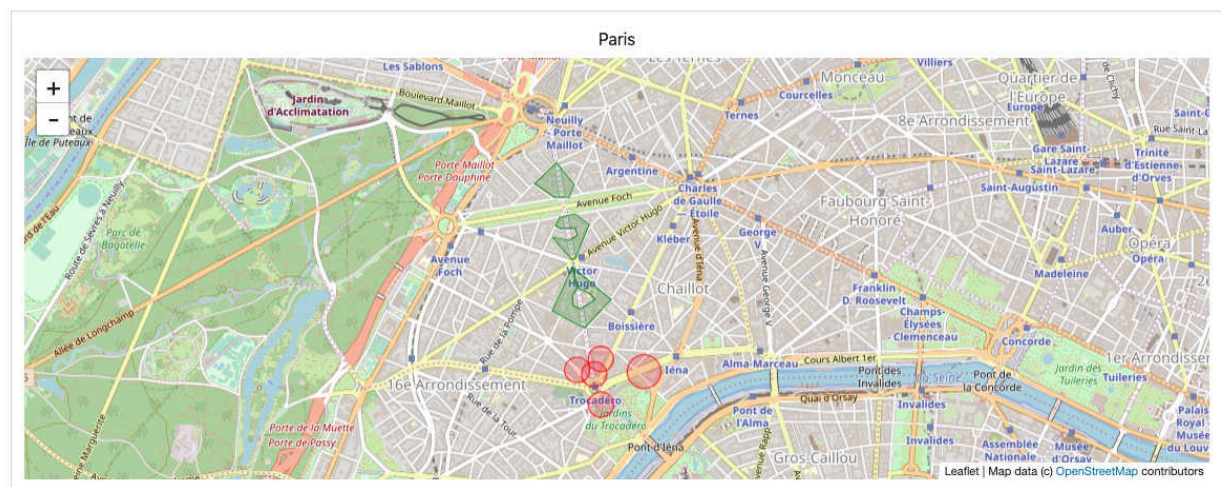
paris =
visuals.createMap(tracker,lat="GPS_latitude",lon="GPS_longitude",size=10,color="speed",color_range=(0,50),
title="Paris",popup=["vehicle","speed"],marker_border=False,
colors=["#e03531","#ff684c","#f0bd27","#51b364","#8ace7e"],
zoom=15,tracking=True,center=(48.875,2.287583))

```

```

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")

```



ゲージ

ゲージを使用して、セグメントに分割されたグラフィックに単一の数値を表示します。



イベントごとに、イベントの指定された数値を示すゲージと、その値が指定された範囲内のどこにあるかを示すポインタが表示されます。イベントのキー値は、現在のゲージ値とともにゲージの上部に表示されます。

単一のゲージの色を指定すると、左端のセグメントに適用されます。残りのセグメントは、指定された色の次第に暗い色合いで色付けされます。または、左から右のセグメントに適用されるカラーパレットを指定できます。

次のようなコードを使用してゲージを作成します。

```
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")
```

表 30 ゲージのパラメーター

パラメーター	説明
<i>datasource</i>	ゲージにフィードするイベントコレクションまたはイベントストリームを指定します。
<i>segments</i>	ゲージに表示するセグメント数を指定します。デフォルト値は 3 です。
<i>value</i>	ゲージに表示する数値を含むフィールド値を指定します。
<i>size</i>	ゲージのページサイズを指定します。デフォルト値は 300 です。

パラメーター	説明
shape	ゲージが占める円の大きさを決定する 40～89 の数値を指定します。デフォルト値は 50 です。
range	ゲージに表示される範囲の開始値と終了値を指定します。デフォルト値は (0,100)です。
columns	表示する列の数を指定します。デフォルト値は 3 です。
color	ゲージの左端のセグメントに使用する色を指定します。残りのセグメントは、この値の暗い色合いで色付けされます。
colors	ゲージに表示するカラーパレットを指定します。
line_width	ゲージの描画に使用される線のピクセル幅を指定します。
title	チャートのタイトルを指定します。

コンパス

コンパスを使用して、ナビゲーションの見出しを表す数値を表示します。



次のようなコードを使用して、コンパスを作成します。

```
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)
```

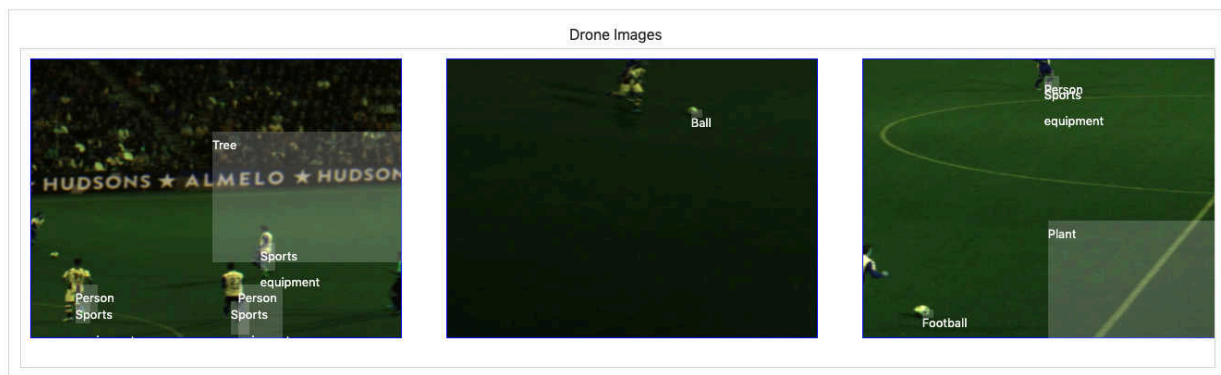
表 31 コンパスのパラメーター

パラメーター	説明
<i>datasource</i>	コンパスにフィードするイベントコレクションまたはイベントストリームを指定します。
heading	コンパスに表示するナビゲーション見出しを含むフィールド値を指定します。
size	ゲージのページサイズを指定します。デフォルト値は 300 です。
columns	表示する列の数を指定します。デフォルト値は 3 です。
heading_color	見出しの矢印に使用する色を指定します。デフォルト値は #89cff0 です。
reciprocal_color	反対向きの矢印の角度に使用する色を指定します。デフォルト値は white です。

パラメーター	説明
bg_color	コンパスの中心に使用する色を指定します。デフォルト値は #f8f8f8 です。
outer_color	コンパスの外側の円に使用する色を指定します。デフォルト値は #89cff0 です。
line_width	ゲージの描画に使用される線のピクセル幅を指定します。デフォルト値は 1 です。
title	チャートのタイトルを指定します。

画像

画像コンポーネントを使用して、イベント画像データを表示できます。



イベントにオブジェクト検出データが存在する場合、画像コンポーネントは、含まれているオブジェクトを示す注釈を画像内に表示します。

このようなコードを使用して画像を表示します。

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

表 32 画像のパラメーター

パラメーター	説明
datasource	画像データをフィードするイベントコレクションまたはイベントストリームを指定します。
image	コンポーネントに表示する画像データを含むフィールド値を指定します。

パラメーター	説明
image_width	各画像の出力幅を指定します。デフォルト値は 300 です。
image_height	各画像の表示高さを指定します。デフォルト値は 300 です。
orientation	画像の方向 vertical (デフォルト)または horizontal を指定します。
image_border	各画像周囲の境界線を指定します。標準の CSS 構文を使用します (border="1px solid blue" など)。
label_color	オブジェクト検出データが画像に存在する場合に表示されるラベルに使用する色を指定します。
title	チャートのタイトルを指定します。

テーブル

テーブル内で選択した数だけイベントフィールドを表示できます。values のセットを指定しない場合、すべてのイベントフィールドが表示されます。

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

次のようなコードを使用して、テーブルを作成します。

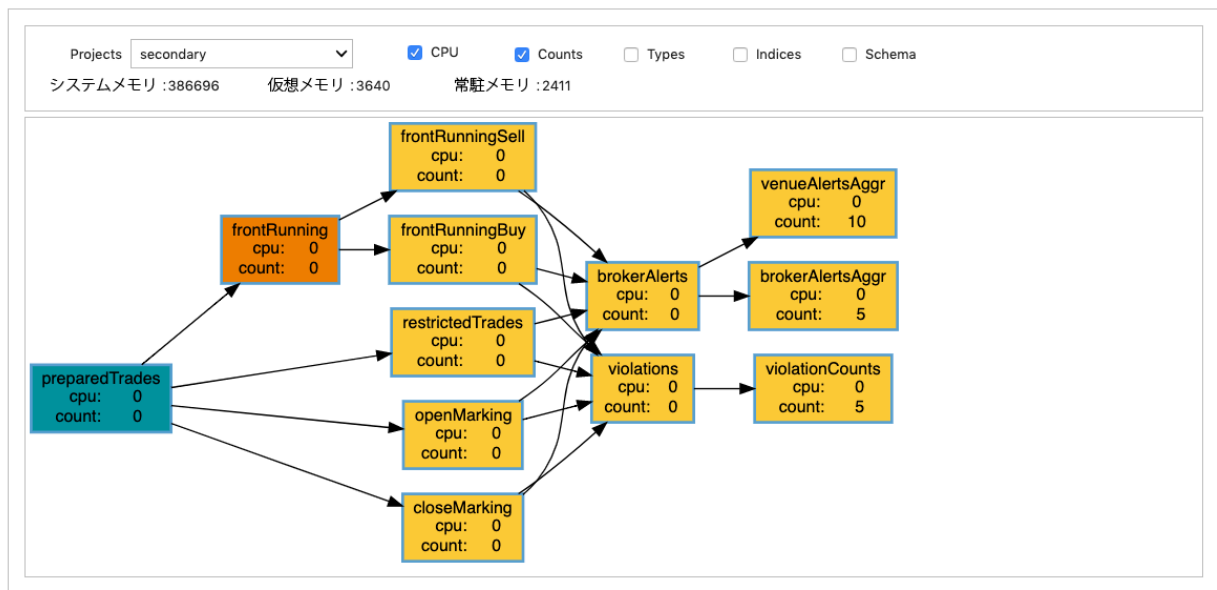
```
table = visuals.createTable(brokerAlerts, values=["brokerName", "total", "restrictedTrades"],
    title="Broker Alerts")
```

表 33 テーブルのパラメーター

パラメーター	説明
datasource	テーブルにフィードするイベントコレクションまたはイベントストリームを指定します。
values	テーブルに表示するイベントフィールドを指定します。
title	チャートのタイトルを指定します。

Event Stream Processing Model Viewer

Model Viewer には、モデルのグラフィック表現が有向グラフの形式で表示されます。各ウィンドウの属性は、選択した数だけ表示できます。一部のモデル属性は動的であり、一部は静的です。Model Viewer には、ESP サーバーのメモリ使用率情報も表示されます。システムメモリ、仮想メモリ、および常駐メモリの情報は、有向グラフの上に表示されます。この情報は、モデルがイベントストリームを処理するときに変化します。



次のようなコードを使用して、Model Viewer を作成します。

```
viewer = visuals.createModelViewer(conn,cpu=True,counts=True,
                                  type=False,index=False,cpu_color="lightblue")
```

表 34 Model Viewer のパラメーター

パラメーター	説明
cpu	ウィンドウの CPU 使用率を表示します。
counts	現在ウィンドウ内にあるイベントの数を表示します。
type	ウィンドウタイプを表示します。
index	ウィンドウのインデックスタイプを表示します。
schema	ウィンドウのインデックスタイプを表示します。
cpu_color	指定された値で始まる色のグラデーションを使用して、CPU 使用率別にモデルノードを色付けします。

デフォルトでは、Model Viewer には ESP サーバーで実行されているすべてのプロジェクトが表示されます。ただし、特定のプロジェクトに集中する場合は、モデルビューアを表示する前にそのプロジェクトを指定します。

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

Event Stream Processing Log Viewer

Log Viewer オブジェクトを使用して、ライブ ESP サーバーログを表示します。メッセージが同時に表示され、コンソールに表示されます。Log Viewer では、最新のログメッセージが上部に表示されます。



次のようなコードを使用して、Log Viewer を作成します。

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

表 35 Log Viewer のパラメーター

パラメーター	説明
<code>conn</code>	接続オブジェクトを指定します。
<code>height</code>	Log Viewer の高さを指定します("500px"など)。デフォルト値は"200px"です。


```

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])

```

そのコードは、次のようなレイアウトを生成します。

