THE POWER TO KNOW.

SAS® Micro Analytic Service 2.1: Programming and Administration Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Micro Analytic Service 2.1: Programming and Administration Guide*. Cary, NC: SAS Institute Inc.

SAS® Micro Analytic Service 2.1: Programming and Administration Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2016

2.1-P1:masag

Contents

About This Book

Audience

This guide is intended for developers and information technology administrators. Developers can use the information to author SAS DS2 or Python code that can be used to process events through SAS Event Stream Processing, which embeds SAS Micro Analytic Service as an internal component. Developers can find information about how SAS Micro Analytic Service processes events, as well as tips, best practices, and restrictions on programming DS2 or Python to run in SAS Micro Analytic Service. Information technology administrators can find information about how to configure SAS Micro Analytic Service and (as an option) Anaconda Python to run inside SAS Event Stream Processing.

Accessibility

For information about the accessibility of any of the products mentioned in this document, see the usage documentation for that product.

Chapter 1

Introduction to SAS Micro Analytic Service

What Is SAS Micro Analytic Service?

SAS Micro Analytic Service 2.1 is a memory-resident, high-performance program execution service that is designed to run inside SAS Event Stream Processing. As a SAS platform service, SAS Micro Analytic Service is not available for individual license but is included in selected SAS solutions. SAS Micro Analytic Service 2.1 is customized for SAS Event Stream Processing. Unlike SAS Micro Analytic Service 1.3, it omits the REST and Java components that support SAS Enterprise Decision Manager. SAS Micro Analytic Service 2.1 includes a core engine that is written in C for high performance and C++ classes that integrate with SAS Event Stream Processing. These capabilities allow both to execute within the same process space for maximum performance. The combination of SAS Event Stream Processing and SAS Micro Analytic Service enables SAS analytics, business logic, and user-written programs to operate on streams of data in motion.

Users of SAS Event Stream Processing can publish SAS analytics, such as predictive models that were created with SAS Enterprise Miner, and they can also author custom programs using the SAS DS2 or Python programming languages. The custom programs execute inside SAS Event Stream Processing applications. SAS Micro Analytic Service can host multiple programs inside SAS Event Stream Processing simultaneously.

SAS Micro Analytic Service supports a subset of the DS2 programming language, which includes language features that are suitable for the high-performance execution of transactions. Specific rules and restrictions are detailed in Chapter 5, “DS2 Programming for Event Stream Processing with SAS Micro Analytic Service,” on page 19.

SAS Micro Analytic Service also supports Anaconda Python 3.4 and Anaconda Python 2.7. Python programs that are written for SAS Micro Analytic Service might include custom functions, and they can use any third-party Python packages that have been deployed to a local Anaconda Python environment. Specific rules for using Python are detailed in Chapter 7, “Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service,” on page 31.

What Is SAS Event Stream Processing?

Event stream processing is a form of complex event processing technology that is often used in data and decision applications. It analyzes and processes large volumes of streaming data quickly, helping you analyze events in motion—even as they are generated. Instead of storing data and running queries against it, the SAS Event Stream Processing Engine stores the queries and streams data through them. In this way, continuous analysis of data occurs as it is received, updating the intelligence as new events occur. Incoming data is read through adapters that are part of a publish-and-subscribe architecture used to read data feeds. Objects written in XML, DS2, Python, and C++ are used to model business logic within the engine. Data that passes through the engine is examined for patterns and can be filtered to more permanent storage. Prebuilt high-performance adapters publish filtered data to other downstream applications. The SAS Event Stream Processing Engine is designed for top performance with a flexible threaded processing model for submillisecond response and high-volume throughput.

The SAS Event Stream Processing Engine uses data flow models to define an ordering of source and derived event windows; the former defines the schema of event streams flowing into the system while the latter determines how these incoming event streams are processed. These data flow models are directed graphs that are frequently referred to as *continuous queries*, and they are the defining characteristic of event stream processing systems. The reason these models are referred to as continuous queries is that event stream processing systems use a set of known (or modeled) queries of interest. Also, they continuously update the resultant set of those queries (also known as *derived windows*) as new events are published to the system. Applications or end users can subscribe to any set of windows (for which they are authorized). Window events can also be queried through SQL in either an ad hoc or on-demand manner.

The SAS Event Stream Processing Engine is event driven- and data flow-centric in that event streams are continuously published to the system and processed through a set of data flow models (or continuous queries). As events are absorbed into the platform and processed though the continuous queries, windows (or nodes in the data flow model) can be queried or subscribed to. Each derived window in the graph is defined by relational operators (for example, join, aggregate, or filter), pattern matching, or procedural processing (for example, an extension to relational processing and pattern matching). However, this is not standard for all event stream processes.

SAS Micro Analytic Service executes inside the event stream processing procedural window, enabling continuous queries to process events with SAS analytics and custom logic that is written in DS2 or Python.

Chapter 2

Concepts

Overview

DS2 and Python programs that are published to SAS Micro Analytic Service, whether user-written or generated by SAS analytical solutions, are known as *modules*. This term reflects the language-neutral nature of SAS Micro Analytic Service interfaces.

A module is a collection of methods. For DS2, a module represents one DS2 package and its methods. For Python, a module is a collection of Python functions.

Module methods can be used to process events in a SAS Event Stream Processing continuous query. The results of such processing create derived events that flow to downstream components in the continuous query. In addition to generating derived events, module methods can influence SAS Event Stream Processing by interrogating and setting event opcodes and flags. Event opcodes and flags are covered in more detail in the DS2 and Python programming chapters that follow. Also, for more information about SAS Event Stream Processing, see the Documentation in the Knowledge Base, at http://support.sas.com.

SAS Micro Analytic Service uses two internal component types to manage the modules that are published to it. These are the module context and the revision. A third component, the user context, provides execution environments containing sets of module contexts. SAS Micro Analytic Service automatically manages user contexts for SAS Event Stream Processing by maintaining one user context per dfESPproject object. Therefore, unlike previous versions of SAS Micro Analytic Service, user contexts do not appear in the user interfaces.

Note: If you plan to write DS2 modules to deploy to SAS Micro Analytic Service, follow the programming guidelines described in Chapter 5, “DS2 Programming for Event Stream Processing with SAS Micro Analytic Service,” on page 19. If you plan to write Python modules to deploy to SAS Micro Analytic Service, follow the programming guidelines described in Chapter 7, “Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service,” on page 31.

Module Context

A module represents program code. In the case of DS2, each module represents exactly one DS2 package. If you are unfamiliar with DS2 packages, see “Understanding DS2 Methods and Packages” in *SAS 9.4 DS2 Language Reference*. Every module is owned by exactly one user context.

In the case of Python, each module represents a Python program, and each module method represents a function in the Python program.

SAS Micro Analytic Service supports module revisions and is capable of hosting and executing multiple revisions of a module concurrently. When SAS Micro Analytic Service compiles a DS2 package or Python program, it creates a revision of that module. Therefore, a module is a container of revisions. It also houses any compiler warning or error messages that were generated from the latest revision compilation or compilation attempt.

Note: SAS Micro Analytic Service 2.1 runs the latest revision of a module by default.

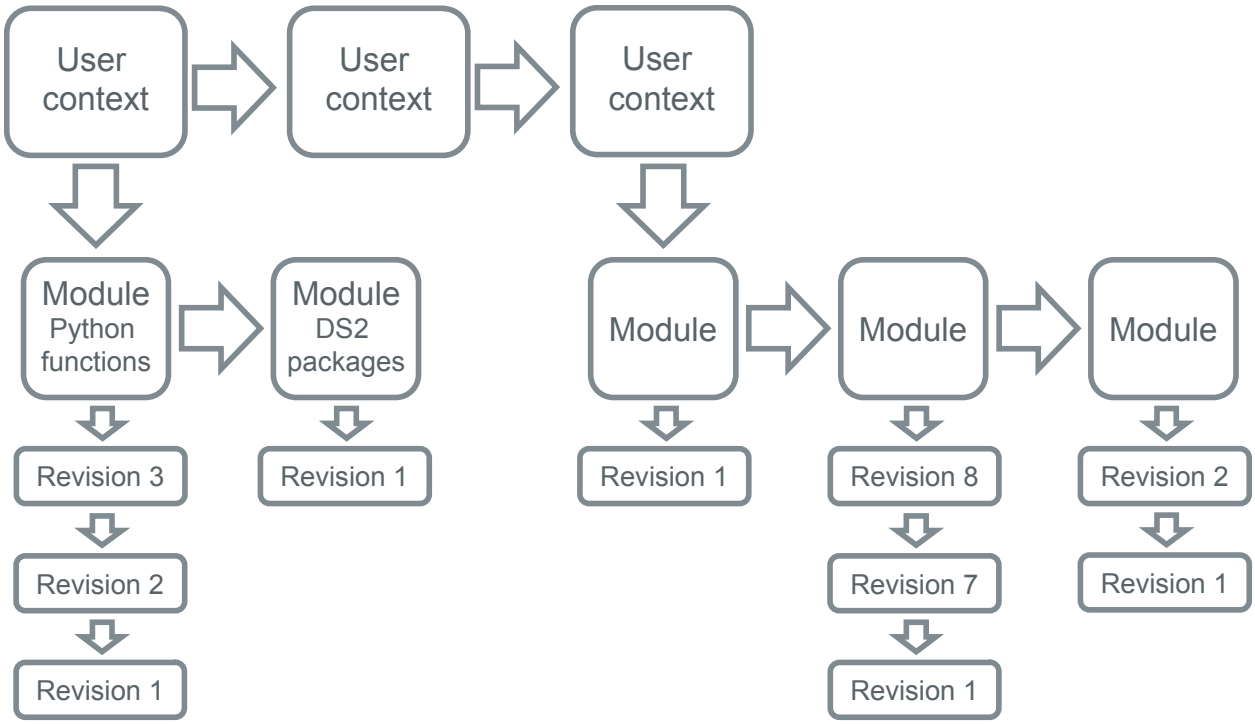
Revision

A revision is a version of a module. Each revision contains source code, an executable code stream (optimized binary executable), and metadata. The metadata describes the methods and method signatures of the module.

SAS Micro Analytic Service assigns a revision number to each revision, which is a monotonically increasing integer value beginning with 1. A revision is uniquely identified by module name and revision number. When you reference a revision, specifying revision number 0 selects the latest revision.

Note: When modules are published to SAS Micro Analytic Service by SAS Event Stream Processing, only the latest revision is retained.

Figure 2.1 *Component Hierarchy*



Chapter 3

Publishing to SAS Micro Analytic Service in SAS Event Stream Processing

Overview

The SAS Event Stream Processing Engine uses data flow models to define an ordering of source and derived event windows. The former defines the schema of event streams flowing into the system, and the latter determines how these incoming event streams are processed. For more information about SAS Event Stream Processing, see the Documentation in the Knowledge Base, at http://support.sas.com.

The derived event window in which SAS Micro Analytic Service operates is the procedural window. Within a continuous query application, procedural windows might be configured to receive events from one or more source windows. These source events can be processed by SAS Micro Analytic Service, which might generate zero or more derived events. These events, in turn, can be subscribed to by downstream windows.

There are two ways to define a SAS Event Stream Processing continuous query: either by creating a continuous query XML definition or by coding a continuous query in C++. Both enable you to publish DS2 and Python modules to SAS Micro Analytic Service. They also specify which of a module's methods are used to process each input source window's events.

Object Hierarchy

SAS Event Stream Processing provides an assortment of elements for use in building continuous query applications, enabling a wide variety of business needs to be met. Some of the elements that incorporate SAS Micro Analytic Service functionality are

described briefly below. For more information, see Documentation in the Knowledge Base, at http://support.sas.com.

At a minimum, each event stream processing application consists of a SAS Event Stream Processing Engine (a container of projects), at least one project (a container of continuous queries), and at least one continuous query that contains windows and the edges that connect them.

Each project maintains a SAS Micro Analytic Service environment that is shared among the continuous queries within the project. In the C++ case, DS2 and Python modules are published to SAS Micro Analytic Service by calling methods on the project. In the XML case, they are published by including the <mas-modules> tag (a sub-element of <project>).

SAS Micro Analytic Service operates on the data that is contained in events received by procedural windows. The procedural window provides a C++ method and an XML element for mapping source window events to module methods. The XML tag is named <mas-map>, which is a sub-element of <window-procedural>.

C++ Methods

C++ continuous query authors can use the following methods for publishing, replacing, or removing DS2 and Python modules and for mapping source window events to module methods.

```
dfESPproject::publishToMAS(    const char *name,
language_t lang,
const char *sourceCode,
const char *description)
```

dfESPproject::publishToMAS() publishes a DS2 or Python module to SAS Micro Analytic Service in SAS Event Stream Processing, where the source code to publish is in a string.

Parameter	Description
name	The name of the module. It is used to refer to the module in any subsequent method calls.
language	Specify either DS2 or Python.
sourceCode	A string containing the DS2 or Python source code of the module. See Chapter 5, "DS2 Programming for Event Stream Processing with SAS Micro Analytic Service," and Chapter 7, "Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service," for information about programming modules for Micro Analytic Service in SAS Event Stream Processing.
description	An optional string that contains a description of the module's purpose. Pass in NULL to omit a description.

```
dfESPproject::publishFileToMAS(    const char *name,
language_t lang,
const char *filePath,
const char *description)
```

dfESPproject::publishFileToMAS() publishes a DS2 or Python module to SAS Micro Analytic Service, where the source code is to be read from a text file.

Parameter	Description
name	The name of the module. It is used to refer to the module in any subsequent method calls.
language	Specify either DS2 or Python.
filePath	The path to the file containing the DS2 or Python source code of the module, including filename and extension. See Chapter 5, "DS2 Programming for Event Stream Processing with SAS Micro Analytic Service," and Chapter 7, "Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service," for Micro Analytic Service in SAS Event Stream Processing.
description	An optional string that contains a description of the module's purpose. Pass in NULL to omit a description.

```
dfESPproject::replaceMASmodule(    const char *name,
language_t lang,
const char *sourceCode,
const char *description)
```

dfESPproject::replaceMASmodule() replaces an existing DS2 or Python module in SAS Micro Analytic Service, where the replacement source code to publish is in a string. If the module to replace, indicated by the name parameter, does not exist, then replaceMASmodule() does not publish the source code.

Parameter	Description
name	The name of the module. It is used to refer to the module in any subsequent method calls.
language	Specify either DS2 or Python.
sourceCode	A string containing the DS2 or Python source code with which to replace the existing code. See Chapter 5, "DS2 Programming for Event Stream Processing with SAS Micro Analytic Service," and Chapter 7, "Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service," for information about programming modules for Micro Analytic Service in SAS Event Stream Processing.
description	An optional string that contains a description of the module's purpose. Pass in NULL to omit a description.

```
dfESPproject::replaceMASmoduleFromFile(    const char *name,
language_t lang,
const char *filePath,
const char *description)
```

dfESPproject:: replaceMASmoduleFromFile () replaces an existing DS2 or Python module in SAS Micro Analytic Service, where the source code is to be read from a text

file. If the module to replace, indicated by the name parameter, does not exist, then replaceMASmoduleFromFile() does not publish the source code.

Parameter	Description
name	The name of the module. It is used to refer to the module in any subsequent method calls.
language	Specify either DS2 or Python.
filePath	The path to the file containing the DS2 or Python source code with which to replace the existing code, including filename and extension. See Chapter 5, "DS2 Programming for Event Stream Processing with SAS Micro Analytic Service," and Chapter 7, "Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service," for Micro Analytic Service in SAS Event Stream Processing.
description	An optional string that contains a description of the module's purpose. Pass in NULL to omit a description.

```
dfESPproject::deleteMASmodule(const char *name,)
```

dfESPproject:: deleteMASmodule() removes the specified module from SAS Micro Analytic Service. The *name parameter is the name of the module to delete.

```
dfESPpcontext::registerMethod_MAS( dfESPwindow *w,
const char *moduleName,
const char *methodName)
```

dfESPpcontext:: registerMethod_MAS () causes the specified source window's events to be processed by the specified module method. The module containing the method must have been published to SAS Micro Analytic Service before calling registerMethod_MAS().

Parameter	Description
w	The source window whose events are to be processed by the specified module method.
moduleName	The name of the module containing the specified method.
methodName	The name of the method to use to process the specified source window's events.

C++ Example

Here is an example C++ continuous query. SAS Micro Analytic Service related method calls are highlighted. Error checking code has been removed to emphasize the main elements of the program, but should always be included in production.

```
// Initialize the SAS Event Stream Processing
// Engine and create a project.
```

```
dfESPengine *myEngine = dfESPengine::initialize(argc,
   argv, "engine", pubsub_DISABLE);
dfESPproject  *project_01 = myEngine->newProject("project_01");

// Publish a DS2 module to SAS Micro Analytic Service,
// in this case a credit risk scoring model.
project_01->publishFileToMAS("credit_risk", DS2,
   "./credit_risk.ds2", "Credit Risk Model");

// Create a continuous query containing source and procedural windows.
dfESPcontquery  *cq_01 = project_01->newContquery("contquery_01");
sw_01 = cq_01->newWindow_source("sourceWindow_01",
dfESPindextypes::pi_HASH,dfESPstring("ID*:int32,custid:int32,
   loanid:int32,amount:money"));
dfESPwindow_procedural *pw_01;
pw_01 = cq_01->newWindow_procedural("proceduralWindow_01",
dfESPindextypes::pi_RBTREE, dfESPstring("ID*:int32,risk_score:double,
   derog_remarks:string"));
dfESPpcontext *ctxt = new dfESPpcontext();

// Register the score() method of the credit_risk module to process
// the source window's events.
ctxt->registerMethod_MAS(sw_01, "credit_risk", "score");

// Add an edge (directed arrow) between the two windows and start the project.
pw_01->registerMethodContext(ctxt);
cq_01->addEdge(sw_01, pw_01);
project_01->setNumThreads(2);
myEngine->startProjects();
```

XML Example

Here is an example of an XML continuous query definition. SAS Micro Analytic Service related elements are highlighted.

```
<engine port='55555'>
  <description>
    This example has one source window and one procedural
    window. The procedural window uses DS2 code to calculate
    a value from the source window.

    The engine element creates the single engine top level container which
    sets up dfESP fundamental services such as licensing and logging.
    This single engine instance wraps one or more projects that wrap
    one or more continuous queries and different types of windows.
  </description>
  <projects>
    <project name='trades_proj' pubsub='auto' threads='4'>
      <description>
        This is to create a project. A project specifies a container
        that holds one or more continuous queries and are backed by a
        thread pool of user defined size. You can specify the pubsub
        port and type, number of threads for the project, index type,
        and a tag token data flow model.
```

```
      </description>
        <mas-modules>
          <mas-module language="ds2" module="module_1" func-names='compute_volume'>
            <description>
              <![CDATA[This is a SAS Micro Analytic Service module in DS2 ]]>
            </description>
            <code>
              <![CDATA[
              ds2_options sas;  /* SAS-style missing value handling */
              package module_1/overwrite=yes;
                method compute_volume(int quantity, double price, in_out int volume);
                  volume = quantity * price;
                end;
              endpackage;
              ]]>
            </code>
            </mas-module>
          </mas-modules>
      <contqueries>
        <contquery name='trades_traders_cq' trace='pw1'>
          <description>
            This specifies the continuous query container that holds
            a collection of windows and enables you to specify the
            connectivity between windows. You can turn on tracing
            for a list of windows, and specify the index type for
            windows in the query.
          </description>
          <windows>
            <window-source name='Trades' index='pi_RBTREE'>
              <description>
                This defines a source window. All event streams must
                enter continuous queries by being published or
                injected into a source window.
              </description>
              <schema>
                <fields>
                  <field name='tradeID' type='string' key='true'/>
                  <field name='security' type='string'/>
                  <field name='quantity' type='int32'/>
                  <field name='price' type='double'/>
                  <field name='traderID' type='int64'/>
                  <field name='time' type='string'/>
                </fields>
              </schema>
              <connectors>
                <connector class='fs' name='pub'>
                  <properties>
                    <property name='type'>pub</property>
                    <property name='fstype'>csv</property>
                    <property name='fsname'>input.csv</property>
                    <property name='transactional'>true</property>
                  </properties>
                </connector>
              </connectors>
            </window-source>
            <window-procedural name='pw1'>
```

```
            <description>
              This defines a procedural window. The window passes
              all fields of the event as variables to the DS2 program.
            </description>
            <schema>
              <fields>
                <field name='tradeID' type='string' key='true'/>
                <field name='security' type='string'/>
                <field name='quantity' type='int32'/>
                <field name='price' type='double'/>
                <field name='traderID' type='int64'/>
                <field name='time' type='string'/>
                <field name='volume' type='int32' key='true'/>
              </fields>
            </schema>
            <mas-map>
              <window-map module="module_1" revision="0" source="Trades"
                 function="compute_volume"/>
            </mas-map>
            <connectors>
              <connector class='fs' name='sub'>
                <properties>
                  <property name='type'>sub</property>
                  <property name='fstype'>csv</property>
                  <property name='fsname'>output.csv</property>
                  <property name='snapshot'>true</property>
                </properties>
              </connector>
            </connectors>
          </window-procedural>
        </windows>
        <edges>
          <description>
            This fully specifies the continuous query with window
            connectivity, which is a directed graph.
          </description>
          <edge source='Trades' target='pw1'/>
        </edges>
      </contquery>
    </contqueries>
    <project-connectors>
      <connector-groups>
        <connector-group name='sub_group'>
          <connector-entry connector='trades_traders_cq/pw1/sub'
              state='running'/>
        </connector-group>
        <connector-group name='pub_group'>
          <connector-entry connector='trades_traders_cq/Trades/pub'
             state='finished'/>
        </connector-group>
      </connector-groups>
      <edges>
        <edge source='sub_group' target='pub_group'/>
      </edges>
    </project-connectors>
  </project>
```

```
  </projects>
</engine>
```

Data Type Mappings

SAS Micro Analytic Service 2.1 executes within the procedural windows of continuous query applications. A continuous query can specify that events from one or more upstream windows be processed by SAS Micro Analytic Service when those events are received by a given procedural window. This specification maps each such input window's events to one of the methods that have been published to SAS Micro Analytic Service.

During continuous query initialization, when a window's events are registered with a method, SAS Micro Analytic Service inspects the window's event schema and automatically maps the event's fields to method input parameters by matching event field names with parameter names. It is legal for some, none, or all of the names to match. It is also legal for methods to have no input parameters. In that case, no input mapping is done. SAS Micro Analytic Service performs similar matching of the specified method's output parameter names with the procedural window's event field names. Therefore, at run time, when an event is received from the specified window, method input values are taken from that event, the method is executed, and the results are used to create a derived event. The event flows downstream from the procedural window to any subscribers.

The following table describes the data type mappings between event field types and DS2 method parameter types, and between event field types and Python function argument types. SAS Micro Analytic Service automatically translates the data types as needed according to the table below. The Event Stream Processing Event Field Type column lists the schema tag of each data type.

Event Stream Processing Event Field Type	Event Stream Processing Type Description	DS2 Method Parameter Type	DS2 Type Description	Python Function Argument Type	Python Type Description
int32	32-bit signed integer	int	32-bit signed integer	int	Signed integer
int64	64-bit signed integer	bigint	64-bit signed integer	long	Long integer
double	IEEE double	double	IEEE double	float	Floating-point real
string	UTF-8 string	char, nchar, varchar, nvarchar	UTF-8 string	string	Unicode string
money	192-bit fixed decimal	double	IEEE double	float	Floating-point real

Event Stream Processing Event Field Type	Event Stream Processing Type Description	DS2 Method Parameter Type	DS2 Type Description	Python Function Argument Type	Python Type Description
date	Date and time, as seconds since January 1, 1970	bigint	Seconds since January 1, 1970	long	Seconds since January 1, 1970
stamp	Date and time, as microseconds since January 1, 1970	bigint	Microseconds since January 1, 1970	long	Microseconds since January 1, 1970

Note: **String** translates to either a single character type or to a variable length string type in DS2, depending on how the DS2 method parameter is declared. Be careful not to pass a multi-character string to a single char argument in DS2, as run-time errors might occur.

Note: The **money** type is presented to DS2 or Python as double or float, respectively.

The SAS Event Stream Processing Engine uses the UNIX epoch for date and time values (January, 1, 1970). The values are presented to DS2 or Python as bigint or long, respectively, using the SAS epoch (January 1, 1960). Native DS2 and Python date and time types are not supported for public arguments. Seconds since 1960 is the SAS datetime value, which makes calling SAS date and time functions convenient in DS2. **Stamp** translates similarly, but with microsecond resolution rather than second resolution.

Chapter 4

Processing Event Opcodes and Flags

Operation Codes and Flags

Each event contains an operation code, or opcode, and a set of flags. For a detailed explanation of these constructs, see *SAS Event Stream Processing: Deployment Guide*. SAS Micro Analytic Service module methods offer you the option of examining a source window event's opcode and flags and setting the opcode and flags of a derived event.

Note: This practice is recommended only for advanced SAS Event Stream Processing users.

These are the opcodes:

- insert
- update
- delete
- upsert
- safedelete

One or more flags can be set in a given event, depending on the event opcode and circumstances. Here are the possible flags:

- N — normal
- P — partial update
- R — retention

DS2 and Python module authors can add zero or more of the following special arguments to their DS2 method or Python function signatures:

_inOpcode
: populated with the source window event opcode when the module method is called. _inOpcode is an input argument and, if included, must appear before any output arguments in the method signature. _inOpcode is a string type, and its value must be **`insert`**, **`update`**, **`delete`**, **`upsert`**, or **`safedelete`** when the method is called.

_outOpcode
: used to either set the opcode of the derived event to emit, or to cause no derived event to be emitted. If _outOpcode is omitted from the method signature, SAS Micro Analytic Service transfers the opcode of the source window event to the derived event. This is the standard behavior under normal circumstances. If _outOpcode is included in the method signature and is set to missing, emission of the derived event is skipped. If _outOpcode is included and is not set to missing, the value of _outOpcode is used to set the opcode of the derived event. The value that is set must be either **insert**, **update**, **delete**, **upsert**, or **safedelete**. To achieve normal processing when _outOpcode is included, the method author must also include _inOpcode and set _outOpcode=inOpcode. _outOpcode is an output argument. Therefore, it must appear after all input arguments in the method signature.

_inFlags
: populated with the source window flags when the module method is called. _inFlags is a string type containing one character per source window event flag that is set. For example, **N** and **NR** are possible values. Reserve space for at least three characters for the _inFlags argument. _inFlags is an input argument. Therefore, if included, it must appear before any output arguments in the method signature.

_outFlags
: used to set the flags of the derived event to emit. If _outFlags is omitted from the method signature, SAS Micro Analytic Service transfers the flags of the source window event to the derived event. This is the standard behavior under normal circumstances. If _outFlags is included and is set to missing, SAS Micro Analytic Service defaults to standard behavior and copies the source window event flags to the derived event.

DS2 Opcodes Example

The following simple example illustrates how to conditionally set the derived event opcode.

In this example, if the source window event's opcode is **delete**, and its quantity is greater than 2000, set the derived event's opcode to **update**, and set the price to 8.50. Otherwise, preserve normal processing by assigning _inOpcode to _outOpcode, and set the price to 10.00.

```
ds2_options sas;
package module_1/overwrite=yes;
method test_function(varchar(16) _inOpcode, int quantity,
  in_out double price, in_out varchar _outOpcode);
    if (_inOpcode = 'delete') and (quantity > 2000) then
    do;
        _outOpcode = 'update';
        price = 8.50;
    end;
    else
    do;
        _outOpcode = _inOpcode;
        price = 10.00;
    end;
end;
endpackage;
```

Chapter 5

DS2 Programming for Event Stream Processing with SAS Micro Analytic Service

Overview

SAS Micro Analytic Service 2.1 supports a subset of the DS2 programming language that is suitable for high-performance transaction processing in real time. This chapter covers only that subset. Note that DS2 batch processing is not supported.

For more information about the DS2 programming language, see *SAS DS2 Language Reference*.

DS2 Source Code Prerequisites

The DS2 source code submitted to SAS Micro Analytic Service should begin with the following statement, just above the PACKAGE statement:

```
"ds2_options sas"
```

. This option instructs DS2 to use SAS missing value handling and helps ensure that your DS2 program behaves the same as if it were run in SAS Foundation. It should end with this statement:

```
"endpackage"
```

The code cannot contain DATA statements, PROC statements, or THREAD statements. The source code should contain one and only one DS2 package, and this package can contain as many methods as desired.

It is a best practice to include a line feed character at the end of each source code line. This line feed character makes it easier to use compiler warning and error messages that include line numbers.

SAS Micro Analytic Service and SAS Foundation

Although DS2 is supported by both SAS Foundation and SAS Micro Analytic Service, SAS Micro Analytic Service has a lightweight, high-performance engine that does not support either the full SAS language or PROC statements. Therefore, PROC statements cannot be used. However, here is an effective DS2 authoring and testing mechanism: develop your DS2 packages in SAS Foundation using PROC DS2 and publish those packages to SAS Micro Analytic Service after removing the surrounding PROC DS2 syntax.

Programming Blocks

Each DS2 module represents exactly one package, and therefore the DS2 PACKAGE statement plays a major role in SAS Micro Analytic Service. A DS2 package contains one or more methods, and methods can contain a wide variety of DS2 language constructs. Package methods work well with rapid transaction processing because they can be called over and over again with little overhead, as transactions flow through the system. By contrast, the DS2 THREAD and TABLE statements are batch-oriented and are not supported.

The following code blocks are supported:

- PACKAGE…ENDPACKAGE
- METHOD…END
- DO…END

The following code blocks are batch-processing oriented and are not supported:

- TABLE…ENDTABLE
- THREAD…ENDTHREAD

Similarly, the following statements are not supported: OUTPUT and SET

- OUTPUT
- SET

Public and Private Methods and Packages

Overview

Public methods are DS2 package methods that can be called by clients that are external to SAS Micro Analytic Service, such as SAS Event Stream Processing.

When a public method is registered with SAS Event Stream Processing as an event processor, the method's arguments are automatically mapped to the fields of the given source window and procedural window events. You register the method either by calling dfESPproject::registerMethod_MAS() or by including it in a window-map entry of an XML project definition.

Note: The method-argument-to-event-field mappings are by name and are case sensitive.

DS2 package methods to be used for event processing must follow all of the public method rules described below.

Private methods and packages are SAS Micro Analytic Service concepts, rather than DS2 features.

SAS Micro Analytic Service can host public DS2 packages and private DS2 packages. Private DS2 packages have fewer restrictions on the DS2 features that can be used than public packages have. Although a private DS2 package cannot be called directly, it can be called by another DS2 package. Private DS2 packages are useful as utility functions, as solution-specific built-in functions, or for solution infrastructure. See your SAS solution documentation for a description of the solution-specific built-in functions that you can use when authoring custom DS2 modules.

A public DS2 package can contain private methods, as long as it contains at least one public method. Any method that does not conform to the rules for public methods is automatically treated as private. Private methods are allowed and do not produce errors if they contain correct DS2 syntax. Private methods are not callable externally. Therefore, they do not show up when querying the list of methods within a package. However, they can be called internally by other DS2 package methods. Here are several typical uses of private methods:

- Small utility functions that return a single, non-void, result.
- Methods containing DS2 package arguments. These are not callable externally.

Public Method Rules

Public methods must conform to the following rules:

- The return type must be void. Rather than using a single return type, public methods can return multiple outputs, where each output argument specifies the in_out keyword in the method declaration. Non-void methods are treated as private.
- Arguments that are passed by reference (meaning ones that specify in_out) are treated as output only. True update arguments are not supported by public methods. This restriction results in more efficient parameter marshaling and supports all interface layers, including REST.

- Input arguments must precede output arguments in the method declaration. It is permissible for a method to have only inputs or only outputs. However, if both are present, all inputs must precede the outputs.
- DS2 packages might not be passed as arguments in public methods. The presence of a DS2 package argument results in the method becoming private.
- The VARARRAY statement might not be present in the argument list of a public method. VARARRAY is a DS2 statement, not a data type. The presence of VARARRAY in a methods argument list causes the method to become private.
- For a full list of data types that can be used as public method arguments, see "Supported DS2 Data Types" on page 24.

Public Method Example

The example below illustrates a valid public method. It has a void return type (no RETURNS clause), uses only publicly supported data types, and treats in_out arguments as output only.

```
method quickSortStep (int lowerIndex, int higherIndex, in_out double numbers[10]);

      dcl int i;
      dcl int j;
      dcl int pivot;
      dcl double temp;

      i = lowerIndex;
      j = higherIndex;

      /* Calculate the pivot number, taking the pivot as the
       * middle index number. */
      pivot = numbers[ceil(lowerIndex+(higherIndex-lowerIndex)/2)];

      /* Divide into two arrays */
      do while (i <= j);
            /**
             * In each iteration, identify a number from the left side that
             * is greater than the pivot value. Also identify a number
             * from the right side that is less than the pivot value.
             * Once the search is done, then exchange both numbers.
             */
            do while (numbers[i] < pivot);
              i = i+1;
            end;
            do while (numbers[j] > pivot);
                j = j-1;
            end;
            if (i <= j) then do;
                temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;

                /* Move the index to the next position on both sides. */
                i = i+1;
                j = j-1;
            end;
```

```
        end;

        /* Call quickSort recursively. */
        if (lowerIndex < j) then do;
            quickSortStep(lowerIndex, j, numbers);
        end;
        if (i < higherIndex) then do;
            quickSortStep(i, higherIndex, numbers);
        end;
    end;
```

Here is another example of a public method that illustrates the use of the HTTP package calling out to a web service using a POST request and then getting a response.

```
method httppost( nvarchar(8192) url,
                 nvarchar(67108864) payload,
                 in_out nvarchar respbody,
                 in_out int hstat, in_out int rc );
  declare package http h();
  rc = h.createPostMethod( url );
  if rc ne 0 then goto Exit;
  rc = h.setRequestContentType( 'application/json;charset=utf-8' );
  if rc ne 0 then goto Exit;
  rc = h.addRequestHeader( 'Accept', 'application/json' );
  if rc ne 0 then goto Exit;
  rc = h.setRequestBodyAsString( payload );
  if rc ne 0 then goto Exit;
  rc = h.executeMethod();
  if rc ne 0 then goto Exit;
  hstat = h.getStatusCode();
  if hstat lt 400 then h.getResponseBodyAsString( respbody, rc );
  else respbody = '';
  Exit:
  h.delete();
end;
```

Private Method Example

The example below generates a private method in SAS Micro Analytic Service. It has a non-void return type. That is, it has a RETURNS clause in the declaration, which specifies a single integer return value.

```
method isNull(double val) returns int;
  return null(val) OR missing(val);
end;
```

Method Overloading

SAS Micro Analytic Service does not support method overloading when it is running in the SAS Event Stream Processing Engine.

CAUTION:

If you publish a DS2 package that contains overloaded methods, run-time errors can occur.

Argument Types Supported in Public Methods

Overview

SAS Micro Analytic Service supports a subset of the DS2 data types for use as public method arguments. Data types in the unsupported list can still be used in the body of a (public or private) DS2 package method, and as arguments to private methods. The lists of publicly supported and unsupported data types are given below.

Note: Any additional types added to the DS2 programming language in future releases should be considered unsupported unless otherwise stated in the SAS Micro Analytic Service documentation.

Supported DS2 Data Types

- BIGINT
- CHAR(n)
- DOUBLE
- INTEGER
- NCHAR(n)
- NVARCHAR(n)
- VARCHAR(n)

Unsupported DS2 Data Types

- BINARY(n)
- DATE
- DECIMAL(p, s)
- NUMERIC(p, s)
- PACKAGE
- TIME(p)
- TIMESTAMP(p)
- TINYINT
- VARBINARY(n)

Chapter 6

Best Practices for DS2 Programming

Overview

This section describes best practices that are recommended when programming in DS2 for any environment. They are not unique to SAS Micro Analytic Service.

Global Packages Versus Local Packages

Overview

The scope of a package instance makes a difference. Package instances that are created in the global scope typically are created and deleted (allocated and freed) once and used over and over again. Package instances that are created in a local scope are created and deleted each time the scope is entered and exited. For example, a package instance that is created in a method's scope is created and deleted each time a method is called. The creation and deletion time can be costly for some packages.

The following examples use the hash package. This technique can be used for all packages.

Example of Optimized Code

This example creates a hash package instance that is global, created and deleted with the package instance, and reused between calls to load_and_clear.

```
/** FAST **/
package mypack;
  dcl double k d;
  dcl package hash h([k], [d]);

  method load_and_clear();
    dcl double i;
    do k = 1 to 100;
      d = 2*k;
      h.add();
    end;
    h.clear();
  end;
endpackage;
```

Example of Poorly Optimized Code

This example creates a hash package instance that is local to the method and created and deleted for each call to load_and_clear.

```
/** SLOW **/
package mypack;
  dcl double k d;

  method load_and_clear();
    dcl package hash h([k], [d]);
    dcl double i;
    do k = 1 to 100;
      d = 2*k;
      h.add();
    end;
    h.clear();
  end;
endpackage;
```

Replacing SCAN (and TRANWRD) with DS2 Code

Consider the following code:

```
i = 1;
onerow = TRANWRD(SCAN(full_table, i, '|'), ';;', ';-;');
do while (onerow ~= '');
    j = 1;
    elt = scan(onerow, j, ';');
    do while (elt ~= '');
        * processing of each element in the row;
        j = j+1;
        elt = SCAN(onerow, j, ';');
```

```
    end;
    i = i+1;
    onerow = TRANWRD(SCAN(full_table, i, '|'), ';;', ';-;');
end;
```

You can make the following observations:

- SCAN consumes adjacent delimiters. Therefore, TRANWRD is required to manipulate each row into a form that can be traversed element by element.
- SCAN starts at the front of the string each time. Therefore, the aggregate cost is O(N^2).
- SCAN and TRANWRD require NCHAR or NVARCHAR input. If full_table is declared as a CHAR or VARCHAR input, it must be converted to NVARCHAR, then processed, and then converted back to VARCHAR in order to be captured into the onerow value.

Here is code that replaces this type of loop with a native DS2 solution and that thus avoids these problems by collecting the necessary details into a package:

```
dcl package STRTOK row_iter();
dcl package STRTOK col_iter();
row_iter.load(full_table, '|');
do while (row_iter.hasmore());
    row_iter.getnext(onerow);
    col_iter.load(onerow, ';');
    do while (col_iter.hasmore());
        col_iter.getnext(elt)
        * processing of each element;
    end;
end;
```

The supporting package, STRTOK, is shown below. It can be used to replace SCAN and TRANWRD pairs anywhere in DS2.

```
/** STRTOK package - extract subsequent tokens from a string.
 * So named because it mirrors (in a safe way) what is done by the original
 * strtok(1) function available in C.
 */
package sasuser.strtok/overwrite=yes;
  dcl varchar(32767) _buffer;
  dcl int strt blen;
  dcl char(1) _delim;

  /* Loads the current object with the supplied buffer and delimiter
   * information. This avoids the cost of constructing and destructing the
   * object, and allows the declaration of a STRTOK outside of the loop in which
   * it is used.
   */
  method load(in_out varchar bufinit, char(1) delim);
    _buffer = bufinit .. delim;
    _delim = delim;
    strt = 1;
    blen = length(_buffer);
  end;

  /* Are there more fields? 1 means there are more fields. 0 means there are
   * no more fields.
   */
```

```
  method hasmore() returns integer;
    if (strt >= blen) then return 0;
    return 1;
  end;

  /* The void-returning GETNEXT method places the next token in the supplied
   * variable, tok.
   */
  method getnext(in_out varchar tok);
    dcl char(1) c;
    dcl int e;
    tok = '';
    if (hasmore()) then do;
      e = strt;
      c = substr(_buffer,e,1);
      do while (c ~= _delim);
        tok = tok .. c;
        e = e + 1;
        c = substr(_buffer,e,1);
      end;
      strt = e + 1;
    end;
  end;

  /* The value-returning GETNEXT method returns the next token. This version is
   * more computationally expensive because it requires an extra copy, as opposed to
   * the void-returning version, above.
   */
  method getnext() returns varchar(32767);
    dcl varchar(32767) tok;
    getnext(tok);
    return tok;
  end;

  /* Construct a STRTOK object using the parameters as initial values.
   */
  method strtok(varchar(32766) bufinit, char(1) delim);
    load(bufinit, delim);
  end;

  /* Construct a STRTOK object without an initial buffer to be consumed.
   */
  method strtok();
    strt = 0; blen = 0;
  end;
endpackage; run;
```

Using STRTOK instead of SCAN and TRANWRD avoids the CHAR to NCHAR conversions and reduces CPU because of how STRTOK retains the intermediate state between calls to the getnext() methods. Therefore, it is O(N) instead of O(N^2).

Hash Package

With both the DATA step and DS2, note the size of the key. A recent program carried out many hash lookups with a 356-byte key. Hashing is an O(1) algorithm; the "1" with the hash package is the length of the key. The longer the key, the longer the hash function takes to operate.

```
dcl char(200) k1 k2;
dcl double d1 d2;

/* If k1 and k2 are always smaller than 200, then    */
/*  size them smaller to reduce the time spent in    */
/*  the hash function when adding and finding values */
/*  in the hash package.                             */
dcl package hash([k1 k2], [d1 d2]);
```

Character-to-Numeric Conversions

When converting a string to a numeric value, note the encoding of the string. When the string is a single-byte encoding, DS2 translates the value to a TKChar (UCS-2 or UCS-4) for conversion. The longer the string, the longer the time it takes to do the conversion.

```
dcl char(512) s;
dcl nchar(512) ns;
dcl double x;
s = '12.345';
ns = '12.345';

x = s;                /* slow */
x = substr(s,1,16);   /* faster */
x = substr(ns,1,16);  /* even faster, avoids transcoding */
```

Passing Character Values to Methods

In SAS Micro Analytic Service, DS2 method input parameters are passed by value. What this means is that a copy of the value is passed to the method. When passing character parameters, a copy of the parameter is made to ensure that the original value is not modified. Making sure that character data is sized appropriately ensures that less copying occurs.

DS2 method output parameters, which are specified by the in_out keyword, are passed by reference. Therefore, no copy is made.

```
method copy_made(char(256) x);
  ...
end;

method no_copy(in_out char x);
```

```
  ...
end;
```

Performing the Computation Once

If a computation is repeated multiple times to compute the same value, you can perform the computation once and save the computed value. For example, the following code block performs the computation, compute(x), four times:

```
if compute(x) > computed_max then computed_max = compute(x);
if compute(x) < computed_min then computed_min = compute(x);
```

If compute(x) always computes the same value for a given value of x, then the code block can be modified to perform the computation once and save the computed value:

```
computed_x = compute(x);
if computed_x > computed_max then computed_max = computed_x;
if computed_x < computed_min then computed_min = computed_x;
```

Moving Invariant Computations Out of Loops

If a computation inside a loop computes the same value for each iteration, improve performance by moving the computation outside the loop. Compute the value once before the loop begins and use the computed value in the loop. For example, in the following code block, compute(x) is evaluated during each iteration of the DO loop:

```
do i = 1 to dim(a);
  if (compute(x) eq a[i]) then ...;
end;
```

If compute(x) is invariant (meaning that it always computes the same value for each iteration of the loop), then the code block can be modified to perform the computation once outside the loop:

```
computed_x = compute(x);
do i = 1 to dim(a);
  if (computed_x eq a[i]) then ...;
end;
```

Chapter 7

Python Programming for SAS Event Stream Processing with SAS Micro Analytic Service

Introduction

SAS Micro Analytic Service 2.1 supports modules that are written in the Python programming language. A Python module represents a Python program, and the module's methods represent the functions within the Python program.

Here is an example of a Python public function that can be hosted by SAS Micro Analytic Service. This example has no output. Input arguments are given in the function's argument list. This example has input variables a and b. Outputs of the function must be listed after "Output:" in the quoted string that follows the function definition. The output variables should match the variables listed in the return statement.

```
def calcATimesB(a, b):
  "Output: "
  print ("Function with no output variables.")
  c = a * b
  print ("Result is: ", c, ", but is not returned")
  return None
```

Note: Input and output argument names live in a single namespace. Therefore, they cannot be the same. This means that in_out arguments are not supported. This is true for all module types in SAS Micro Analytic Service. This is not an issue in Python, as a new variable can be assigned the value of an input argument and then safely added to the output list. If the "Output:" line is missing, the function is not exposed as a callable function through SAS Micro Analytic Service. However, that function can be called internally. As a result, any function without the "Output:" line is a private function. The func0() function is an example of such a private function. SAS Micro Analytic Service parses the code to create a dictionary of the methods and their signatures.

Public and Private Methods

Private and public methods are SAS Micro Analytic Service concepts, rather than Python features. Any method having the "Output:" doc string is considered a public method. If a method does not have the "Output:" doc string, then it is considered a private method. SAS Micro Analytic Service can host public and private Python methods, where a method is a Python function. Although a private method cannot be called directly, it can be called by another method (public or private). Private methods are useful as utility functions. Private methods are not callable externally. Therefore, they do not show up when querying the list of methods within a package. However, they can be called internally by other methods.

Python modules can be published containing all public methods, or a mixture of public and private methods. Both public and private methods can call other functions that either exist within the module internally or in external Python packages, including third-party libraries.

All public functions returning more than one output argument must return a tuple containing all of the output arguments. This can be done by returning all of the arguments separated by commas. When returning zero arguments from a public function you are still required to include the "Output:" doc string to indicate a public function. It should simply be "Output:", with no output arguments listed. You can omit the return statement, return "None", or return an empty tuple.

An example of returning an empty tuple is **`return ()`**. An example of returning "None" is **`return None`**. One output argument can be returned as is. It is not required to be returned within a tuple. Here is an example: **`return a`**.

Therefore, it could be **`return a,b,c or return (a,b,c)`**.

Note: Order does matter. Therefore, the order in the return statement must match the order in the "Output:" line. A best practice is to cut and paste from one to the other.

Example

The following example illustrates the use of each SAS Event Stream Processing data type as input to and as output from a public Python function.

```
#
#    Name: scalarsTest.py
# Purpose: Test the Python program for scalar types
#
# Inputs (name)         (type)
#       inString        String
#       inBool          Boolean
#       inLong          Long
#       inDouble        Double
#       inTimestamp     Long microseconds since 1960
#       inDatetime      Long seconds since 1960
#       inMoney         Double
#
# Outputs (name)        (type)
```

```
#       outString       String
#       outBool         Boolean
#       outLong         Long
#       outDouble       Double
#       outTimestamp    Long microseconds since 1960
#       outDatetime     Long seconds since 1960
#       outMoney        Double
#
# Note: Event stream processing presents the timestamp as
#       long microseconds since 1960 and datetime as long
#       seconds since 1960.

# Import the datetime module to perform datetime operations.
import datetime

def scalarsTest(inString, inBool, inLong, inDouble,
     inTimestamp, inDatetime, inMoney):
    "Output: outString, outBool, outLong, outDouble,
      outTimestamp, outDatetime, outMoney"

    if inString == None:
        outString = None
    else:
        # Convert the casing of the string input.
        outString = inString.swapcase()
    print ("\n inString=", inString, " outString=",
           outString, " (reverse case)")

    if inBool == None:
        outBool = None
    else:
        # Reverse value of the Boolean.
        outBool = not inBool
    print ("\n inBool=", inBool, " outBool=", outBool,
           " (not inBool)")

    if inLong == None:
        outLong = None
    else:
        # Add 10 to long.
        outLong = inLong + 10
    print ("\n inLong=", inLong, " outLong=", outLong,
           " (add 10)")

    if inDouble == None:
        outDouble = None
    else:
        # Add 10.1 to the double.
        outDouble = inDouble + 10.1
    print ("\n inDouble=", inDouble, " outDouble=", outDouble,
           " (add 10.1)")

    if inTimestamp == None:
        outTimestamp = None
    else:
        # Since this is defined as a stamp in event stream processing
```

```
    # schema, this number is long microseconds since 1960.
    # Add one second == 1000000 microseconds.
    outTimestamp = inTimestamp + 1000000
print ("\n inTimestamp=", inTimestamp, " outTimestamp=",
       outTimestamp, " (add one second)")

if inDatetime == None:
    outDatetime = None
else:
    # Since this is defined as date in the event stream processing schema,
    # this number is long seconds since 1960.
    # Add one day.
    outDatetime = inDatetime + (3600 * 24)
print ("\n inDatetime=", inDatetime, " outDatetime=", outDatetime,
       " (add one day)")

if inMoney == None:
    outMoney = None
else:
    # Add 25 cents.
    outMoney = inMoney + 0.25
print ("\n inMoney=", inMoney, " outMoney=", outMoney,
       "(add 25 cents)")


# Return all of the outputs.
return outString, outBool, outLong, outDouble, outTimestamp,
 outDatetime, outMoney
```

Chapter 8

Administration and Deployment

SAS Micro Analytic Service Logging

SAS Micro Analytic Service uses the SAS 9.4 Logging Facility. For more information, see *SAS Logging: Configuration and Programming Reference*. SAS Event Stream Processing provides a default logging configuration file, and that file specifies loggers and appenders in addition to those described in this chapter. For more information about SAS Event Stream Processing, see Documentation in the Knowledge Base, at http://support.sas.com.

SAS Micro Analytic Service uses three loggers named App.tk.MAS, App.tk.MAS.Python, and App.tk.MAS.CodeGen. Code that is hosted by SAS Micro Analytic Service, or the functions that it calls, can use additional loggers.

The logger App.tk.MAS is used for logging various aspects of SAS Micro Analytic Service operation. App.tk.MAS.CodeGen is used for code compilation and generation logging events. App.tk.MAS.Python is used for Python related logging. Normal operations, such as start-up and shutdown, are logged at the INFO level. Detailed information about such operations as compilation start and finish, and others, are logged at the DEBUG level. Warning and error conditions are logged at the WARN or ERROR levels, as appropriate. By default, App.tk.MAS is set to the ERROR level.

App.tk.MAS.CodeGen is used for logging compiler-generated messages, such as compilation warnings and errors. Compiler messages are also retrieved and logged by SAS Event Stream Processing whenever it publishes a module to SAS Micro Analytic Service.

Deployment

Deploying SAS Micro Analytic Service

SAS Micro Analytic Service is deployed automatically when SAS Event Stream Processing is deployed. However, if Python modules are to be used, then Python requires the additional deployment and configuration steps found in "Configuring Python".

Configuring Python

Python 2.7 and 3.4 on 64-Bit Linux

1. Download Anaconda for Linux, Python 3.5, and Linux 64-bit installer from https://www.continuum.io/downloads.
2. After downloading the installer, enter the following in a terminal window. Provide the Python version that you installed. In the following example, Python 3.5 is used.

   ```
   bash Anaconda3-2.5.0-Linux-x86_64.sh
   ```

   Answer yes to the question, **Do you wish the installer to prepend the Anaconda3 install location to PATH in your .bashrc?** These instructions assume that you have used the location **`/users/myuserid/anaconda3`**.
3. Create a Python 3.4 environment by entering the following (note that there are two hyphens before **`name`**). Provide the appropriate Python version.

   ```
   bash
   conda create --name python34 python=3.4
   ```
4. Activate the environment (provide the appropriate Python version).

   ```
   source activate python34
   ```
5. Prepend the Python environment's **`lib`** directory to the LD_LIBRARY_PATH environment variable. Provide the Python version that you installed. In the following example, Python 3.4 is used.

```
LD_LIBRARY_PATH=/users/myuserid/anaconda3/envs/python34/lib:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH%:}
```

Note: Regarding 64-bit Linux, the **`conda create`** and **`source activate`** commands must be run from a bash or zsh shell.

Python 2.*x* uses ASCII as the default encoding. Therefore, you must specify another encoding at the top of the file to use non-ASCII Unicode characters in literals. As a best practice, when using Python 2.*x*, always use the following as the first line of your Python script:

```
# -*- coding: utf-8 -*-
```

Also, in Python 2.*x*, the Unicode literal must be preceded by the letter u. Therefore, literal strings should be written using the following form:

```
u"xxxxx"
```

Note: Python 3.*x* uses UTF-8 as the default encoding. Therefore, these issues affect Python 2.*x* only. When using Python 3.*x*, you can use the default encoding, and you can simply enclose literals in quotation marks.

Further Considerations for Configuring Python

The Anaconda documentation states that Python 3.4 can be run from an Anaconda 2.7 installation by creating and activating a Python 3.4 environment. You cannot do this with embedded Python. Therefore, it is recommended that you use the Python 3.5 installer for both Python 2.7 and 3.4.

When starting SAS Event Stream Processing, do so from a shell in which you have activated Python, thus allowing the process to use Python.

A rich set of Python packages is available, covering a wide variety of computing needs. You might want to add some of these packages to your Python environment.

When you add packages to an Anaconda environment, the packages are placed in **<*your-environment-path*>/lib/python3.4/site-packages**. In order to use the Python scripts that these packages require, add their locations to the PYTHONPATH environment variable.

Note: The use of the configuration scripts espenv and espenv_print, when running Python is highly recommended. These scripts are described in "Configuration Helper Scripts".

If your Python script imports your own .py files, you also must add their location to PYTHONPATH. An example location might be .(dot).

Some packages include a lib directory, which also needs to be added to PYTHONPATH.

Finally, you must add **<*your-environment-path*>/lib/python3.4** to PYTHONPATH.

Anaconda sets the environment variable CONDA_PREFIX when you activate an environment and sets it to the location where Anaconda stores any new Python packages that you install (for example, the site-packages folder).

Here is an example of the locations that you might set for PYTHONPATH, after adding packages to your Python 3.4 environment for 64-bit Linux:

```
export SITE_PACKAGES=$CONDA_PREFIX/lib/python3.4/site-packages
export PYTHONPATH=.:$CONDA_PREFIX/lib/python3.4
export PYTHONPATH=$PYTHONPATH:$SITE_PACKAGES
export PYTHONPATH=$PYTHONPATH:$SITE_PACKAGES/numpy
export PYTHONPATH=$PYTHONPATH:$SITE_PACKAGES/numpy/lib
```

Configuration Helper Scripts

SAS Event Stream Processing includes two shell scripts to assist with configuring environment variables for running SAS Event Stream Processing. Using these scripts to run Python modules in SAS Micro Analytic Service is optional.

The script espenv_print prints a list of shell commands that can be cut and pasted into your .bashrc file or other shell script of your choice. The following is example output from espenv_print:

```
[develop]$ ../package-scripts/espenv_print -i sas -m 3.4
```

```
# The setting environment for an official SAS installation.
# The path name must point to the espenv_print location.
# -i indicates that the content must be stored in SAS or that it is SAS software.
# -m is required only if you plan to use Python, and it is followed
#  by the version number.
#
#
# Save or restore the previous PATH.
#
if [ ! -z ${ESPENV_PATH} ] ; then
   PATH=$ESPENV_PATH
else
   ESPENV_PATH=$PATH
fi

#
# Unset event stream processing specific variables.
#
unset    TKPATH
unset     ESP_I

#
# Set basic variables for event stream processing.
#
DFESP_HOME="/mnt/data/home/userid/work/esp"
ESP_I="-I /mnt/data/home/userid/work/esp/include/esptk
       -I /mnt/data/home/userid/work/esp/include/mas"

LD_LIBRARY_PATH="/mnt/data/home/userid/work/esp/lib:
                /mnt/data/home/userid/SASFoundation/sasexe"
           PATH="/mnt/data/home/userid/work/esp/bin:
                 /mnt/data/tools/java/jdk1.8.0_60/bin:
                 /mnt/data/tools/apache/apache-ant-1.8.2/bin:
                 /mnt/data/tools/maven/apache-maven-3.3.9/bin:
                 /mnt/data/tools/doxy/1.8.8/bin:
                 /mnt/data/tools/git/bin:
                 /opt/teradata/client/14.10/tbuild/bin:
                 /usr/lib64/qt-3.3/bin:/opt/rh/python27/root/usr/bin:
                 /usr/local/bin:/bin:/usr/bin:/usr/local/sbin:
                 /usr/sbin:/sbin:/mnt/data/home/userid/work/esp/develop/bin"

#
# Set the basic variables for event stream processing,
# SAS Micro Analytic Service, or Python.
#

source activate python34

PYTHONPATH=".:$CONDA_PREFIX/lib/python3.4:
            $CONDA_PREFIX/lib/python3.4/site-packages:
            $CONDA_PREFIX/lib/python3.4/site-packages/pandas:
            $CONDA_PREFIX/lib/python3.4/site-packages/numpy:
            $CONDA_PREFIX/lib/python3.4/site-packages/numpy/lib"

LD_LIBRARY_PATH="$CONDA_PREFIX/lib:$LD_LIBRARY_PATH"
```

```
export DFESP_HOME TKPATH ESP_I LD_LIBRARY_PATH PATH PYTHONPATH ESPENV_PATH
```

The script espenv can generate the same shell commands as espenv_print. However, instead of printing the commands to the console, it executes the commands, configuring the current shell to run Python modules in SAS Event Stream Processing.

To make Python configuration as convenient as possible, add the espenv function to your .bashrc file using the following steps. This makes the function available to call from any folder, as long as you are in a bash shell. Use the espenv function if you switch back and forth between Python 2.7 and Python 3.4. If you plan to use only one of the versions, add the output of espenv_print to your .bashrc file instead. This makes the configuration run automatically whenever you launch a bash shell.

1. Open espenv_func and copy the code.
2. Change directories to your home directory.
3. Paste the code into your .bashrc file.
4. Copy espenv_print to your home directory.
5. Edit espenv_print and search for the following lines:

```
#*********************************************************************
# MODIFY PYTHONPATH (BELOW) WHENEVER PYTHON PACKAGES HAVE BEEN ADDED   *
# OR REMOVED FROM YOUR ANACONDA PYTHON 3.4 ENVIRONMENT                *
#*********************************************************************
PYTHONPATH='.:$CP/lib/python3.4:$CP/lib/python3.4/site-packages:$CP/lib/
             python3.4/site-packages/numpy:$CP/lib/python3.4/site-packages
             /numpy/lib:$CP/lib/python3.4/site-packages/sklearn'
```

6. The script contains two lines that set PYTHONPATH, one for Python 3.4 and the other for Python 2.7. Edit the appropriate PYTHONPATH and add paths for each Python package that you have installed and intend to use. Be sure to separate each path with a colon. $CP represents the root location where packages are stored in your Anaconda environment (such as the folder called site-packages). espenv_print automatically retrieves this location from Anaconda when the script is executed.

   Note: Some Python packages require you to store more than one path because they store Python modules and executables in more than one folder.

7. Save your changes.

Note: Here is an alternative method:

1. Run espenv_print.
2. Cut and paste the output into your .bashrc file.
3. Append :$PATH to the end of the line that starts with PATH.
4. Append the following to the end of LD_LIBRARY_PATH:

   ```
   :$LD_LIBRARY_PATH
   ```

For a complete description of these scripts and their options, see *SAS Event Stream Processing: Deployment Guide*.

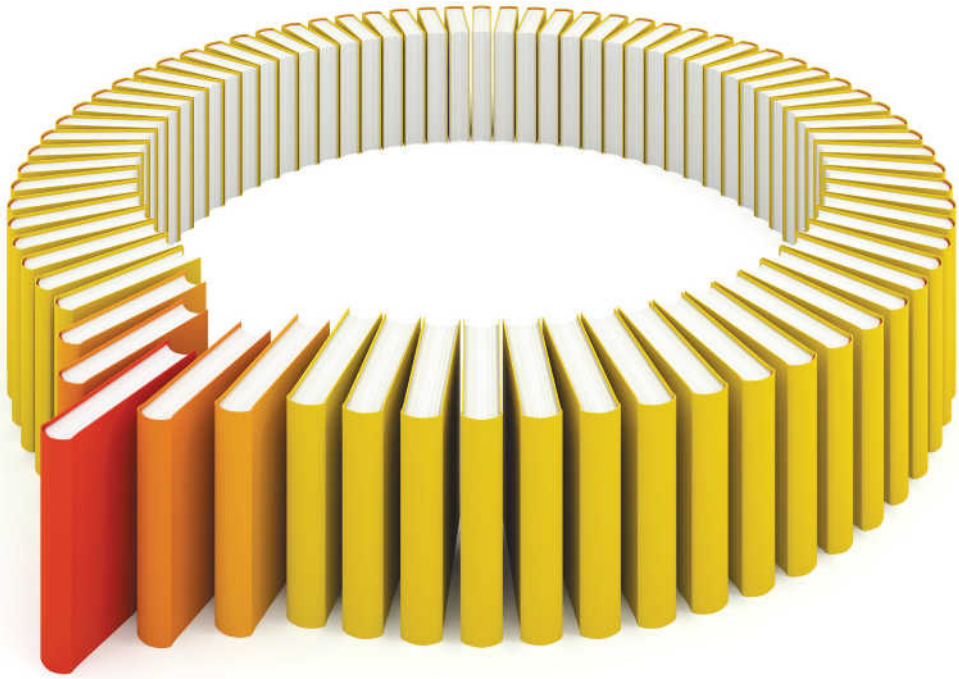
Recommended Reading

- *Encryption in SAS 9.4*
- *SAS 9.4 DS2 Language Reference*
- *SAS 9.4 Logging: Configuration and Programming Reference*
- *SAS Event Stream Processing: Overview*
- *SAS Event Stream Processing: Deployment Guide*
- *SAS Event Stream Processing: Examples*

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Index



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.