SAS® Micro Analytic Service 5.3: Programming and Administration Guide

SAS® Documentation
September 9, 2019

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® Micro Analytic Service 5.3: Programming and Administration Guide*. Cary, NC: SAS Institute Inc.

SAS® Micro Analytic Service 5.3: Programming and Administration Guide

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

September 2019

5.3-P1:masag

Contents

About This Book . . . . . *vii*
What's New in SAS Micro Analytic Service 5.3 . . . . . *ix*
Accessibility . . . . . *xiii*

PART 1 Understanding SAS Micro Analytic Service 1

Chapter 1 • Introduction to SAS Micro Analytic Service **3**
What Is SAS Micro Analytic Service? 3

Chapter 2 • Concepts **5**
Overview 5
User or Business Context 6
Module Context 6
Revision 7
Architecture 8

PART 2 Using SAS Micro Analytic Service with SAS Event Stream Processing 9

Chapter 3 • Publishing to SAS Micro Analytic Service in SAS Event Stream Processing **11**
Overview 11
Object Hierarchy 12
XML Example 12
Data Type Mappings 15

Chapter 4 • Generating Derived Events **19**
Processing Event Opcodes and Flags 19
Derived Event Suppression and NULL Key Fields 21
Generating Multiple Derived Events from a Single Source Event 21
Multiple Derived Events and NULL Key Fields 23

Chapter 5 • DS2 Programming for SAS Micro Analytic Service **25**
Overview 26
DS2 Source Code Prerequisites 26
DS2 Identifiers 26
SAS Micro Analytic Service and SAS Foundation 27
Programming Blocks 27
Restrictions When Working with DS2 and SAS Micro Analytic Service 28
Public and Private Methods and Packages 28
Argument Types Supported in Public Methods 31
Implicit Data Type Conversions 32
Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service 33
Performing Calls between SAS Micro Analytic Service Modules 33
Using Analytic Store Models 35

Chapter 6 • State Sharing between Modules **39**
Overview 39
Shared Vectors 40
Shared Hash Tables 46

Chapter 7 • Best Practices for DS2 Programming in SAS Event Stream Processing **51**
Overview 51
Return Results 51
Global Packages versus Local Packages 52
Replacing SCAN (and TRANWRD) with DS2 Code 53
Hash Package 55
Character-to-Numeric Conversions 55
Data Type Conversions 55
Passing Character Values to Methods 56
Performing the Computation Once 56
Moving Invariant Computations Out of Loops 56

Chapter 8 • Python Support in SAS Micro Analytic Service **59**
Introduction 59
Example 60
Public and Private Methods 62
Working with Python and SAS Micro Analytic Service 65
Configuring Python for SAS Event Stream Processing 66

Chapter 9 • SAS Micro Analytic Service Logging and Deployment **69**
SAS Micro Analytic Service Logging 69
Deployment 70

PART 3 Using SAS Micro Analytic Service with SAS Intelligent Decisioning or SAS Model Manager 73

Chapter 10 • DS2 Programming for SAS Micro Analytic Service **75**
Overview 76
DS2 Source Code Prerequisites 76
DS2 Identifiers 76
SAS Micro Analytic Service and SAS Foundation 77
Programming Blocks 78
Restrictions When Working with DS2 and SAS Micro Analytic Service 78
Public and Private Modules and Methods 79
Argument Types Supported in Public Methods 82
Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service 84
Performing Calls between SAS Micro Analytic Service Modules 84
Managing Large DS2 Modules 86
Composite Modules 89
Referencing Modules and Composite Submodules 89
Using Analytic Store Models 89

Chapter 11 • State Sharing between Modules **93**
Overview 93
Shared Vectors 94
Shared Hash Tables 100

Chapter 12 • Best Practices for DS2 Programming in SAS Intelligent Decisioning **105**
Overview 105
Return Results 105
Global Packages versus Local Packages 106
Replacing SCAN (and TRANWRD) with DS2 Code 107
Hash Package 109
Character-to-Numeric Conversions 109
Passing Character Values to Methods 109
Performing the Computation Once 110
Moving Invariant Computations Out of Loops 110

Chapter 13 • Python Support in SAS Micro Analytic Service **111**
Introduction 111
About Creating Python Modules 112
Public and Private Methods 113
Working with Python and SAS Micro Analytic Service 116
Compiling Python Modules 117
Configuring Python for SAS Intelligent Decisioning 118

Chapter 14 • Administration **121**
SAS Micro Analytic Service Administration 121
Database Access with DS2 122
Starting and Stopping SAS Micro Analytic Service 124
Synchronous, Asynchronous, and Timed Execution 124
SAS Micro Analytic Service Configuration 125
SAS Micro Analytic Service Logging 131
SAS Micro Analytic Service Security and Authorization 133
Secure DS2 HTTP Package Usage 134
Moving Objects by Using the SAS Viya Transfer Service 134

PART 4 Appendixes 137

Appendix 1 • Executing Python Modules in DS2 Modules **139**
DS2 Interface to Python 139
Sample DS2 Module Operations 141
Configuring Support for a DS2 PyMAS Package 146

Appendix 2 • SAS Micro Analytic Service Return Codes **149**

Appendix 3 • REST Server Error Messages and Resolutions **161**

Appendix 4 • Table Service Driver Reference **167**
DB2 Driver Reference 167
FedSQL Driver Reference 173
ODBC Driver Reference 176
Oracle Reference 183
PostgreSQL Driver Reference 189
SAS Data Set Reference 194
Teradata Reference 198

Appendix 5 • SAS Micro Analytic Service Tuning Guidelines 203

Appendix 6 • Applying a New License 205

Recommended Reading 207
Index 209

About This Book

Audience

SAS Intelligent Decisioning, SAS Model Manager, and SAS Event Stream Processing include SAS Micro Analytic Service, which enables you to publish SAS analytics, business rules, and user-written modules into operational environments. In addition, a variety of SAS analytics is available to you, and you can author custom logic in DS2 or Python, as well as deploy a combination of the module types that are specified above.

This guide is intended for developers and information technology administrators who use a SAS Event Stream Processing, SAS Intelligent Decisioning, or SAS Model Manager environment. Because aspects of SAS Micro Analytic Service differ according to environment, ensure that you refer to the section that applies to your environment.

Included here is information about how SAS Micro Analytic Service processes transactions and events, as well as tips, best practices, and restrictions on programming DS2 or Python to run in SAS Micro Analytic Service.

Technology administrators can find information about how to configure SAS Micro Analytic Service. Also included is information about how to configure Python and SAS Micro Analytic Service to run Python code (optional).

What's New in SAS Micro Analytic Service 5.3

Overview

SAS Micro Analytic Service is a memory-resident, high-performance program execution service that is included in selected SAS solutions. It provides hosting for DS2 and Python programs and supports a "compile-once, execute-many-times" usage pattern. In addition to supporting a rich variety of SAS analytics and business rules, SAS Micro Analytic Service enables you to author DS2 or Python code that is customized to your specific needs.

SAS Micro Analytic Service 5.3 includes the following enhancements:

- No restrictions on Python version
- Ability to perform calls between SAS Micro Analytic Service modules
- Support for direct Python module usage
- External credential functionality for compiling and executing Python modules
- Asynchronous module execution
- Asynchronous module publication
- Resource management improvements
- Batch recording for the SAS Intelligent Decisioning subject-contact service
- SAS lockdown support for controlling Python execution

No Restrictions on Python Version

SAS Micro Analytic Service no longer requires the Anaconda distribution of Python 3.4 or 2.7. You can now use any version of Python.

Perform Calls between SAS Micro Analytic Service Modules

You can use the DS2 MASCall package to enable separate DS2 modules, published to SAS Micro Analytic Service, to call one another across separate module executables.

Support for Direct Python Module Usage

Using the SAS Micro Analytic Service REST interface, you can create a Python module directly without using any DS2 code. This is useful when you need SAS Micro Analytic Service to execute only Python code.

Compile and Execute Python Modules Using Alternate Credentials

For security reasons, you might need to limit the access of Python processes to the file system or other resources of the host server. You can accomplish this by using SAS Viya external credentials functionality. This is supported only when you are using the SAS Micro Analytic Service REST interface.

Execute Modules Asynchronously

By default, module execution is processed synchronously. Using the REST interface, you can now perform asynchronous execution. When you use asynchronous execution, the input parameter values are passed to the server and the execution occurs in a separate thread.

Publish Modules Asynchronously

Using the SAS Micro Analytic Service REST interface, the jobs endpoint enables the caller to submit a Create request or an Update request that is longer than the SAS Viya web server time-out value.

Resource Management Improvements

The SAS Micro Analytic Service REST interface includes several features to support and manage large DS2 modules. This enables a system administrator to balance the size and complexity of the modules against system memory, system responsiveness, and availability.

Batch Recording for the SAS Intelligent Decisioning Subject-Contact Service

The SAS Micro Analytic Service REST interface enables asynchronous recording of contact and response history through the SAS Intelligent Decisioning subject-contact service. This means that the code to record history in the database does not execute in the time frame of executing the decision.

Database access is typically a high-latency activity. By removing this activity from the decision flow, SAS Micro Analytic Service is able to meet strict response time requirements.

SAS Lockdown Support for Controlling Python Execution

If SAS is in a locked-down state, the SAS Micro Analytic Service Python interface is prohibited by default. You can use arguments in the LOCKDOWN statement to enable access to the Python interface.

Accessibility

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

Part 1

Understanding SAS Micro Analytic Service

Chapter 1
Introduction to SAS Micro Analytic Service *3*

Chapter 2
Concepts *5*

Chapter 1

Introduction to SAS Micro Analytic Service

What Is SAS Micro Analytic Service? **3**
Overview 3
About Using SAS Micro Analytic Service 3

What Is SAS Micro Analytic Service?

Overview

SAS Micro Analytic Service is a memory-resident, high-performance program execution service. As a SAS platform service, it is not available for individual license, but is included in selected SAS solutions. SAS Micro Analytic Service provides hosting for DS2 and Python programs and supports a “compile-once, execute-many-times” usage pattern. SAS Micro Analytic Service is multi-threaded and can be clustered for high availability. It can host multiple programs simultaneously, as well as multiple user or business contexts that are isolated from one another.

SAS Micro Analytic Service contains a core engine that is written in C for high performance and, when deployed as part of SAS Event Stream Processing, includes 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.

About Using SAS Micro Analytic Service

SAS Micro Analytic Service is integrated with SAS Event Stream Processing and deployed with SAS Intelligent Decisioning and SAS Model Manager. Note the following information about using SAS Micro Analytic Service with these solutions:

- When used in a SAS Intelligent Decisioning environment, SAS Micro Analytic Service is called as a web application with a REST interface. The REST interface (known as the SAS Micro Analytic Score service) provides easy integration with client applications and adds persistence and clustering for scalability and high availability.

  SAS Intelligent Decisioning generates DS2 programs that implement user-created rule sets and rule flows. It can combine SAS analytics, such as score code generated

by SAS Enterprise Miner, with business rules in order to form decision logic. SAS Micro Analytic Service is used to compile and execute the generated code. For more information, see *SAS Intelligent Decisioning: User's Guide*.

- When SAS Model Manager is installed, a publishing destination is created automatically for SAS Micro Analytic Service. Users can publish models to the SAS Micro Analytic Service publishing destination and then score them within the publishing destination. For more information, see *SAS Model Manager: User's Guide*.
- Users of SAS Model Studio can publish to SAS Micro Analytic Service if SAS Model Manager is also installed.
- Users of SAS Event Stream Processing or SAS Intelligent Decisioning can publish SAS analytics, such as predictive models that were created with a variety of SAS products and analytical procedures. They can also author custom programs using the SAS DS2 or Python programming languages or in the SAS Intelligent Decisioning web application. The custom programs execute inside SAS Event Stream Processing applications. SAS Micro Analytic Service can host multiple programs simultaneously.

  For information about SAS Event Stream Processing, see the product documentation at http://support.sas.com.

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.

SAS Intelligent Decisioning generates DS2 programs that implement user-created rule sets and rule flows. It can combine SAS analytics, such as score code generated by SAS Enterprise Miner, with business rules in order to form decision logic. SAS Micro Analytic Service is used to compile and execute the generated code.

SAS Micro Analytic Service supports the Python programming language. Python programs that are written for SAS Micro Analytic Service might include custom functions. They can use any third-party Python packages that have been deployed to a local Python environment.

Chapter 2

Concepts

Overview 5
User or Business Context 6
Module Context 6
Revision 7
Architecture 8

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 for a wide variety of other purposes, including computing scores, processing data, or making business decisions.

For SAS Event Stream Processing, module methods can be used to process events in a 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.

For SAS Intelligent Decisioning and SAS Model Manager, module methods can be used to automate data-driven decisions. This is accomplished by executing analytical models and business rules against the latest data from online channels, combined with data from operational databases and other data sources.

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 isolated execution environments that contain sets of module contexts and revisions. SAS Micro Analytic Service automatically manages user and module contexts for the user.

SAS Micro Analytic Service automatically manages user contexts for SAS Event Stream Processing by maintaining one user context per event stream processing object.

The SAS Micro Analytic Service REST service, used by SAS Intelligent Decisioning and SAS Model Manager, automatically creates and manages one user context per tenant.

Before writing modules to deploy to SAS Micro Analytic Service, see the following information:

- For DS2 modules, see the programming guidelines for your environment:
  - SAS Event Stream Processing: Chapter 5, "DS2 Programming for SAS Micro Analytic Service," on page 25
  - SAS Intelligent Decisioning and SAS Model Manager: Chapter 10, "DS2 Programming for SAS Micro Analytic Service," on page 75
- For Python modules, see the programming guidelines for your environment:
  - SAS Event Stream Processing: Chapter 8, "Python Support in SAS Micro Analytic Service," on page 59
  - SAS Intelligent Decisioning and SAS Model Manager: Chapter 13, "Python Support in SAS Micro Analytic Service," on page 111

User or Business Context

A *context* is a container for the programs that SAS Micro Analytic Service executes. It is also an isolated execution environment. That is, programs executing in one context are not visible to any other context. Therefore, contexts can be used to provide a separate environment for each user or different business unit, or for any other usage requiring isolation. As noted in the previous section, programs that are hosted by SAS Micro Analytic Service are known as modules. A context is a container of modules.

Because business context and user context are interchangeable terms that describe the two common uses of this single component, this document uses the term *user context* for simplicity.

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 DS2 Language Reference*. Every module is owned by exactly one user context.

In the case of Python, each module represents a collection of related Python functions, and each module method represents one of those functions.

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 or Python module, it creates a revision of that module. Therefore, a module context is a container of revisions. A module context also houses any compiler warning or error messages that were generated from the latest compilation or compilation attempt.

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

Note: The Micro Analytic Service REST interface supports running only the latest revision of a module.

Revision

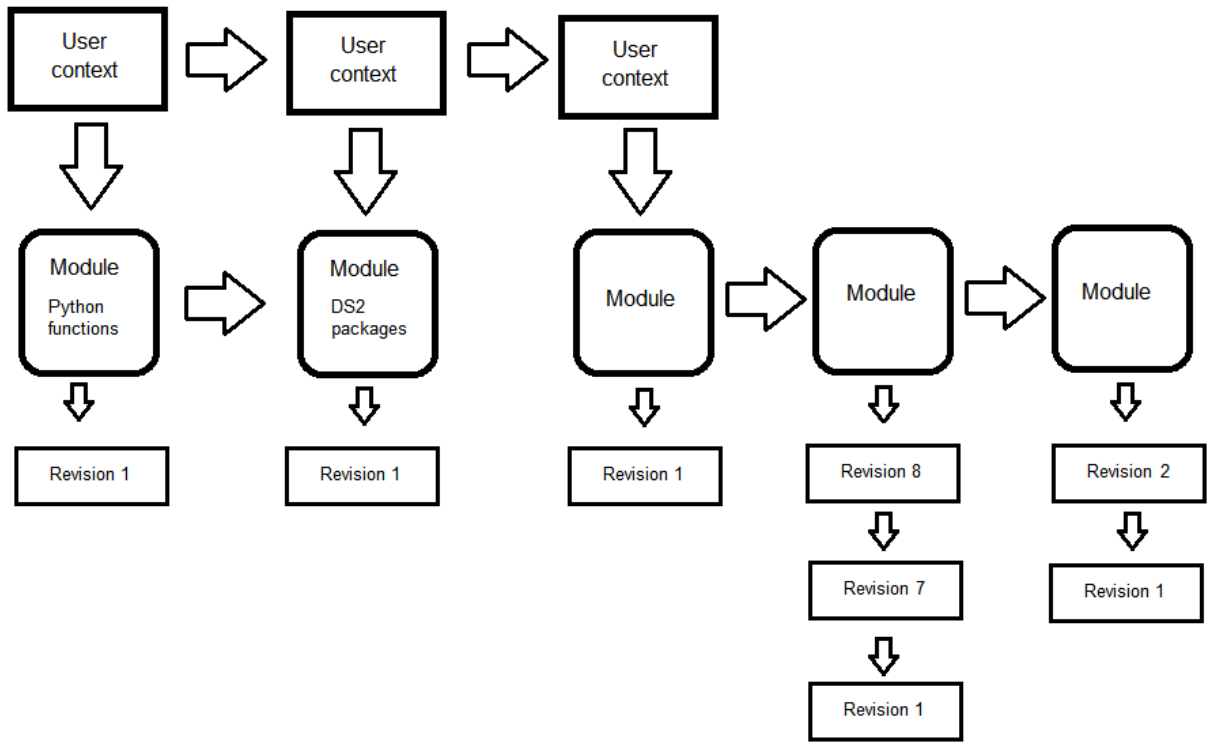
A revision is a version of a module. Each revision contains source code, an executable code stream (optimized binary executable), and metadata. The metadata describes the methods and method signatures of the module.

Revisions provide several advantages, including the ability to roll back to a previous version of a module.

SAS Micro Analytic Service assigns a revision number to each revision, which is a monotonically increasing integer 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*



Note: Revisions can be added and deleted, which might result in non-sequential revision numbers. In the figure above, this is illustrated by one of the module's revisions going from Revision 1 to Revision 7.

Architecture

SAS Micro Analytic Service has a layered architecture:

Core Engine
: The SAS Micro Analytic Service core engine is written in C and is multi-threaded for high performance.

  Note: SAS Event Stream Processing integrates with SAS Micro Analytic Service via a private C interface to the core engine. This interface allows SAS Event Stream Processing to call SAS Micro Analytic Service directly, in-process, for maximum performance. When SAS Micro Analytic Service is deployed with SAS Event Stream Processing, Java and REST layers are omitted.

Java Layer
: a thin Java layer communicates with the core engine through the Java Native Interface (JNI). Commands from the REST interface are passed to the core engine through this Java layer.

REST
: adds functionality such as persistence and clustering support.

  Note: SAS Intelligent Decisioning and SAS Model Manager interface with SAS Micro Analytic Service via REST, and uses all three layers of the architecture.

Part 2

Using SAS Micro Analytic Service with SAS Event Stream Processing

Chapter 3
Publishing to SAS Micro Analytic Service in SAS Event Stream Processing *11*

Chapter 4
Generating Derived Events *19*

Chapter 5
DS2 Programming for SAS Micro Analytic Service *25*

Chapter 6
State Sharing between Modules *39*

Chapter 7
Best Practices for DS2 Programming in SAS Event Stream Processing *51*

Chapter 8
Python Support in SAS Micro Analytic Service *59*

Chapter 9
SAS Micro Analytic Service Logging and Deployment *69*

Chapter 3

Publishing to SAS Micro Analytic Service in SAS Event Stream Processing

Overview 11
Object Hierarchy 12
XML Example 12
Data Type Mappings 15

Overview

SAS Event Stream Processing uses data flow models, known as continuous queries, that define how events are routed among the various windows that make up a continuous query.

Source windows are required for each continuous query. All event streams enter continuous queries by being published or injected into a Source window. Source windows are typically connected to one or more derived windows. Derived windows can detect patterns in the data, transform the data, aggregate the data, analyze the data, or perform computations based on the data. For more information, see the SAS Event Stream Processing documentation at http://support.sas.com.

The derived window in which SAS Micro Analytic Service operates is the Calculate window. Within a continuous query application, Calculate windows can be configured to receive events from one or more source windows. These source events can be processed by modules published to SAS Micro Analytic Service, which can in turn generate zero or more derived events. These derived events can be subscribed to by downstream windows. For more information, see "Working with SAS Micro Analytic Service Modules" in *SAS Event Stream Processing: Using Streaming Analytics*.

SAS Event Stream Processing Studio can be used to author continuous queries. Continuous queries can include elements that publish modules to SAS Micro Analytic Service, and which identify the methods to use to process specific event streams. Up to one method can be specified for each window that streams events directly into a Calculate window (for example, for each upstream window connected to a Calculate window by an edge).

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 the SAS Event Stream Processing documentation 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. DS2 and Python modules are published to SAS Micro Analytic Service 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 Calculate windows. The Calculate window provides 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-calculate>.

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 calculate
    window. The calculate 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='cw1'>
          <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-calculate name='cw1' algorithm='MAS'>
              <description>
                This defines a calculate 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-calculate>
              </windows>
              <edges>
                <description>
                  This fully specifies the continuous query with window
                  connectivity, which is a directed graph.
                </description>
                <edge source='Trades' target='cw1'/ role='data'>
              </edges>
            </contquery>
          </contqueries>
          <project-connectors>
            <connector-groups>
              <connector-group name='sub_group'>
                <connector-entry connector='trades_traders_cq/cw1/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 executes within the Calculate 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 Calculate 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. This is done 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 Calculate 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 Calculate 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
Boolean	8-bit signed	bool	Boolean	bool	Boolean
int32	32-bit signed integer	int	32-bit signed integer	long	Long 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
array(i32)	32-bit signed integer array	int **`parameter`**[]	32-bit signed integer array	list	List of longs
array(i64)	64-bit signed integer array	bigint **`parameter`**[]	64-bit signed integer array	list	List of longs
array(dbl)	IEEE double array	double **`parameter`**[]	IEEE double array	list	List of floating point real
rstring	UTF-8 string	char nchar varchar nvarchar	UTF-8 string	string	Unicode string
blob	Binary large object	binary	Fixed-length binary data	Not supported	Not supported

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.

For the base data types (string, int32, int64, double, datetime, timestamp, and money), data is stored inline in the event. This allows for fast indexing and serialization.

For the array(i32), array(i64), array(dbl), blob, and rstring data types, data is not stored in an event, but rather in another location in memory. The event contains a pointer to the actual data. All of these object data types are reference counted at the object level. This allows an object to be referenced in multiple events, which saves memory and the

amount of time that it would take to create a new object and copy the data. However, note that these data types cannot be used as key fields for an event.

Chapter 4

Generating Derived Events

Processing Event Opcodes and Flags **19**
Overview 19
Operation Codes and Flags 19
DS2 Opcodes Example 21

Derived Event Suppression and NULL Key Fields **21**

Generating Multiple Derived Events from a Single Source Event **21**
Overview 21
Unique Keys 22
Setting Opcodes and Flags on Multiple Derived Events 23

Multiple Derived Events and NULL Key Fields **23**

Processing Event Opcodes and Flags

Overview

As discussed in Chapter 3, "Publishing to SAS Micro Analytic Service in SAS Event Stream Processing," SAS Micro Analytic Service is embedded in the Calculate windows of continuous query applications. One or more upstream windows are connected to a Calculate window by edges. Events *flow* from upstream windows to the Calculate window along these edges.

Modules, which have been published to SAS Micro Analytic Service, process these events. A module is a collection of methods, and each such upstream window can be bound to a specific method. When an event flows from an upstream window to the Calculate window, the method that is associated with that specific upstream window is called. SAS Micro Analytic Service executes the method using input data from the event, and then generates one or more derived events that contain the results of the method execution.

Methods can be used to perform a wide variety of tasks such as scoring, advanced analysis, and real-time decision making.

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: Overview*. 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;
```

Derived Event Suppression and NULL Key Fields

In a DS2 method, when output arguments that are mapped to derived event key fields are set to NULL or MISSING, the following rules (in order of precedence) are observed:

- If all key fields are NULL, the derived event is silently suppressed.
- If some, but not all, of the key fields are NULL, a warning is issued indicating that the derived event is being suppressed because of an incomplete key set.
- If the _outOpcode pseudo argument is present but NULL, the derived event is silently suppressed.

Generating Multiple Derived Events from a Single Source Event

Overview

The methods that are published to SAS Micro Analytic Service can be used to generate multiple derived events when a single source event is received. This is accomplished by coding an array output argument for each derived event field. The values in the arrays are then used to populate the fields of multiple derived events.

A SAS Micro Analytic Service module method can produce one or more arrays. When such a method is mapped to a SAS Event Stream Processing Source window, and when at least one of the output array names matches a derived event field name, multiple derived events can be generated. The number of elements in the matching array or arrays at run time determines the number of events that are generated. As is the case with scalar outputs, any array output that does not match a derived event field name is ignored.

When you are authoring a method that is capable of generating more than one event, the best practice is to produce parallel arrays of the same size. When you follow this practice, each array element contributes one value to each of the derived events, in order. The arrays can contain different numbers of elements across method calls (for example, they might generate one derived event for one call, three derived events for another, and so on). However, for a single method call, the best practice is to produce the same number of elements in each array.

If you ignore these best practices, you can still generate multiple events, but keep the following rules in mind:

- The longest array that maps to a derived event field determines the number of events that are generated.
- When the longest array is being determined, trailing missing values are not counted.
- If an array is shorter than the number of events to be generated, missing values are set in the corresponding field of the derived events for which the array has no data.

Scalar output values, if any, are repeated in the corresponding fields of every derived event.

Unique Keys

SAS Event Stream Processing requires that every event have a unique key composed of one or more event fields. Therefore, the source event's key cannot be duplicated in multiple generated derived events. In that case, SAS Event Stream Processing halts processing and returns a duplicate key error. Because of this, the method author must ensure that each derived event has a unique key value. To do that, produce an array containing unique key values and mark the matching event field as a key in the derived event schema. This technique is useful whenever you author your own module.

SAS Micro Analytic Service can execute modules that are generated by SAS analytics products, where the analytic functions do not produce arrays of unique key values. To accommodate such functions, SAS Micro Analytic Service provides a key generation feature. To use this feature, add a key field named _masRowNum of type int32 to the derived event schema, making it part of a composite key. When _masRowNum is present, SAS Micro Analytic Service populates the field with the row number of each derived event, in order, starting with 1. This feature ensures that the composite key value is unique across multiple generated derived events.

Here is an example of a derived event schema:

```
ID*:int32,_masRowNum*:int32,symbol:int32,
   quantity:int32,price:double,total:double
```

Note: The module method does not need to produce a SAS Event Stream Processing key value when the meta field _masRowNum is present.

Setting Opcodes and Flags on Multiple Derived Events

When a set of derived events is generated from a single source event, you can explicitly set opcodes or flags in the derived events either individually or as a group. To set individual opcodes, use the _outOpcodeArray meta argument. Similar to _outOpcode, _outOpcodeArray can also be used to suppress the generation of any of the individual derived events.

To set flags in the derived events individually, use the _outFlagsArray meta argument.

To set the same opcode in all the derived events that are generated from a single source event, use the _outOpcode meta argument and omit the _outOpcodeArray meta argument. Similarly, to set the same flags in all the derived events that are generated from a single source event, use the _outFlags meta argument and omit the _outFlagsArray meta argument.

For information about the meta arguments _outOpcodeArray, _outFlagsArray, _outOpcode, and _outFlags, see "Operation Codes and Flags" on page 19.

Multiple Derived Events and NULL Key Fields

In a DS2 method that generates multiple derived events from a single input event, when output arguments that are mapped to derived event key fields are set to NULL or MISSING, the following rules (in order of precedence) are observed:

- For any given derived event of a set, if all key fields are NULL, the derived event is silently suppressed.
- For any given derived event of a set, if some of the key fields are NULL, but some are not, a warning is issued indicating that the derived event is being suppressed because of an incomplete key set.
- For any given derived event of a set, if _outOpcodeArray is present but the corresponding _outOpcodeArray value is NULL or MISSING, the corresponding derived event is suppressed.
- If _outOpcodeArray is present and the entire array is NULL or MISSING, the entire derived event set is suppressed.

Chapter 5

DS2 Programming for SAS Micro Analytic Service

Overview **26**

DS2 Source Code Prerequisites **26**

DS2 Identifiers **26**

SAS Micro Analytic Service and SAS Foundation **27**

Programming Blocks **27**

Restrictions When Working with DS2 and SAS Micro Analytic Service **28**
Character Restrictions 28
User-Defined Formats 28

Public and Private Methods and Packages **28**
Overview 28
Public Method Rules 29
Public Method Example 29
Private Method Example 31
Method Overloading 31

Argument Types Supported in Public Methods **31**
Overview 31
Supported DS2 Data Types 31
Unsupported DS2 Data Types 32

Implicit Data Type Conversions **32**

Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service . . . **33**

Performing Calls between SAS Micro Analytic Service Modules **33**
Overview 33
MASCall Methods 33
Examples 34

Using Analytic Store Models **35**
About Analytic Store Models 35
Publishing an Analytic Store Model 36
Calling Analytic Store Models Using DS2 36
Example 36
Configuring ASTORE File System Paths 37
Composite Modules 37

Overview

SAS Micro Analytic Service 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 statement 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. DS2 source code 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.

Note: DS2 supports only a specific style of comment. Comments start with the characters /*, and they end with the characters */. All characters between the starting and ending characters are part of the comment text. Comments can be nested. When there is ambiguity in determining a token, the compiler always chooses the longest possible sequence of characters that can make up a token.

DS2 Identifiers

For DS2 method, package, and argument names, SAS Micro Analytic Service supports regular identifiers and delimited identifiers. When you are using a delimited identifier, any character is allowed, including multi-byte and non-ASCII characters. You must begin and end delimited identifiers with double quotation marks. For complete information, see “DS2 Identifiers” in *SAS DS2 Programmer’s Guide*.

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.

Here is an example PROC DS2 step that illustrates the above mechanism:

```
proc ds2;

ds2_options sas;
package myPackage/overwrite=yes;
method copyArray(char(12) in_array[4], in_out char(12) out_array[4]);
    out_array := in_array;
end;
endpackage;
run;

table _null_;
method init();
   dcl package myPackage p();
   dcl char(12) inarr[4];
   dcl char(12) outarr[4];
   inarr[1] = 'one';
   inarr[2] = 'two';
   inarr[3] = 'three';

   p.copyArray(inarr, outarr);
   put outarr[1]=;
   put outarr[2]=;
   put outarr[3]=;
end;
run;

quit;
```

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

Restrictions When Working with DS2 and SAS Micro Analytic Service

Character Restrictions

The following characters are not allowed in module IDs, package names, public method names, and submodule names:

- backslash (\)
- forward slash (/)
- period (.)
- semicolon (;)

User-Defined Formats

SAS Micro Analytic Service does not support the use of SAS sessions. Because a SAS session is required to create and access user-defined formats, SAS Micro Analytic Service does not support the use of user-defined formats.

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 Calculate 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 82.

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. The DS2 programming language does support method overloading for programs running in other environments, but not when running in SAS Micro Analytic Service.

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 included 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

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

Unsupported DS2 Data Types

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

Implicit Data Type Conversions

When the data types in the event schema differ from the data types in the corresponding method arguments, certain implicit data conversions are supported. The following table contains the supported data type conversions. A conversion is supported to and from each type.

Data Type in Schema	Module Method Argument Data Type
64-bit integer	• 32-bit integer • double • Boolean
32-bit integer	• 64-bit integer • double • Boolean
double	• 32-bit integer • 64-bit integer • Boolean

SAS Micro Analytic Service confirms that a 64-bit integer value can successfully be converted to a 32-bit integer value without overflow, but not all conversions are tested. If

problems are encountered, including those that are outside the SAS Micro Analytic Service domain, an error occurs. For more information about numerical representation, see the topics in Numerical Accuracy in SAS Software.

Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service

The DS2 function inmas() discovers whether SAS Micro Analytic Service is running in the current process and, if so, determines whether the current thread is a member of the SAS Micro Analytic Service worker thread pool. If it is, then the DS2 code is running inside SAS Micro Analytic Service.

The function returns 1 (TRUE) if the DS2 code is executing in SAS Micro Analytic Service, and 0 (FALSE) otherwise.

This can be useful to know when, for example, you have DS2 code that works in various locations, but not in SAS Micro Analytic Service.

Performing Calls between SAS Micro Analytic Service Modules

Overview

When a DS2 module references another DS2 package, the DS2 compiler does the following:

1. Copies, from the source code repository, the source code of the referenced package into the source code of the module to be published. The code is copied inline.

   Note: The current DS2 language infrastructure supports source code repositories only.

2. Compiles the combined source code into a single executable code stream.

A drawback to this approach is that the DS2 package code is repeated in every module that references it. This results in increased memory usage for every redundant copy of a package and longer compilation times.

SAS Micro Analytic Service resolves these issues via the DS2 MASCall package. This package contains methods that enable separate DS2 modules, published to SAS Micro Analytic Service, to call one another across separate module executables.

In addition to a smaller memory footprint and shorter compilation times, this functionality enables a library of modules to be reused by many higher-level modules without penalty.

MASCall Methods

Here are the package MASCall methods:

- allocParms(module_name, method_name)

  This method creates a parameter list for the latest revision of the specified method.

- allocRevParms(module_name, revision_number, method_name)

  This method is similar to allocParms, but it enables you to specify a revision number.

Important: The called module name is case-sensitive on UNIX systems. It is not case-sensitive on Windows systems.

After allocParms() or allocRevParms() is called, the parameter setter methods operate on the newly created parameter list.

Note that for both the setter and getter methods, **`arg_index`** is zero-based. The argument takes a numeric index value, not a method parameter name.

Here are the setter methods:

- Scalar argument setters are of the form:

  return_code = set<type>(arg_index, value)

- Array argument setters are of the form:

  return_code = set<type>Array(arg_index, array_value)

After the input values are set, you can execute the target method by making the following call, which calls the external module method that was previously specified by allocParms() or allocRevParms():

callModule()

Note: Because the method to execute was previously specified, callModule() has no parameters.

Output values returned by the method execution can be retrieved using the MASCall getter methods, which are as follows:

- Scalar argument getters are of the form:

  value = get<type>(arg_index)

- Array argument getters are of the following form.

  get<type>Array(arg_index, array_value, rc)

  Note: DS2 passes arrays and output values by reference.

After the output values are retrieved, call releaseParms() to release the parameters and other resources used for the method. This releases the memory resources used for memory execution and prevents memory leaks.

Examples

Here are DS2 MASCall package example method calls, where **`mc`** is the DS2 MASCall instance variable that is created by calling the package constructor, which takes no arguments. Here are two ways to construct a MASCall package instance named **`mc`**:

- Example 1:

```
declare package mascall mc();
```

- Example 2:

```
declare package mascall mc;
mc = _new_ mascall();
```

General example:

```
rc = mc.allocParms( module_name, method_name );
rc = mc.setDouble( 0, additionalDiscount );
```

```
rc = mc.setString( 1, currentPhone );
rc = mc.setString( 2, currentPlan );
rc = mc.callModule();
treatments = mc.getString( 3 );
mc.releaseParms();
```

The complete set of DS2 package methods follows, where **rc** is the integer return code, and **mc** is the package instance. Note that **arg_index** is zero-based. The argument takes a numeric index value, not a method parameter name.

Parameter resource management:

```
rc = mc.allocParms(    module_name, method_name );
rc = mc.allocRevParms( module_name, revision, method_name );
     mc.releaseParms();
```

Scalar argument setters:

```
rc = mc.setString(   arg_index, value );
rc = mc.setLong(     arg_index, value );
rc = mc.setInt(      arg_index, value );
rc = mc.setDouble(   arg_index, value );
```

Array argument setters:

```
rc = mc.setStringArray(   arg_index, string_array    );
rc = mc.setLongArray(     arg_index, bigint_array    );
rc = mc.setIntArray(      arg_index, integer_array   );
rc = mc.setDoubleArray(   arg_index, double_array    );
```

Execute or call a module's method:

```
rc = mc.callModule( );
```

Scalar argument getters:

```
string_var =    mc.getString(   arg_index );
long_var =      mc.getLong(     arg_index );
int_var =       mc.getInt(      arg_index );
double_var =    mc.getDouble(   arg_index );
```

Array argument getters:

```
mc.getStringArray(   arg_index, string_array,    rc );
mc.getLongArray(     arg_index, bigint_array,    rc );
mc.getIntArray(      arg_index, integer_array,   rc );
mc.getDoubleArray(   arg_index, double_array,    rc );
```

Using Analytic Store Models

About Analytic Store Models

An analytic store file, called an ASTORE file, is a system that allows the state of a trained analytical model to be saved in a transportable form. This enables it to subsequently be used to score new data in a variety of environments. Many SAS analytical procedures save the results from the training phase of model development as analytic store models. A key feature of an ASTORE file is that it can be easily transported from one platform to another. When an analytic store is published to SAS

Micro Analytic Service, the state of the predictive model is restored and is available for scoring new data.

Publishing an Analytic Store Model

Unlike DS2 and Python modules, analytic store models are not published to SAS Micro Analytic Service as source code. Instead, analytic store models consist of binary code and metadata. Client applications deliver analytic store models to SAS Micro Analytic Service as ASTORE disk files.

Note: For information about calling an analytic store model by a DS2 module, see the next section.

Calling Analytic Store Models Using DS2

If an analytic store model has been registered with SAS Micro Analytic Service, it can be called by a DS2 module.

A DS2 module that calls an analytic store model must include an init() method that invokes the score package's setvars() and setkey() methods.

Note: Failure to set this option can cause the system to stop responding on module deletion or on shutdown.

The setvars() method is used by the DS2 score package to map variables to the analytic store model's input and output parameters. The setkey() method takes a SHA-1 hexadecimal key as input and uses it to look up the analytic store model.

SAS Micro Analytic Service automatically calls the init() method, if present, when a DS2 module is published.

Example

```
ds2_options sas;
package astoretest/overwrite=yes;
dcl package score sc();
dcl double CLAGE;
dcl double CLNO;
dcl double DEBTINC;
dcl double DELINQ;
dcl double NINQ;
dcl double VALUE;
dcl double _P_;
dcl double P__EVENT_0;
dcl double P__EVENT_1;
dcl nchar(32) I__EVENT_;
dcl nchar(4) _WARN_;
varlist allvars [_all_];

method init();
   sc.setvars(allvars);
   sc.setkey(n'EB3D1CA20AA0CB74465D25EEE2290E13692AF750');
end;

method preCode();
   _P_ = 0.999;
```

```
end;

method postCode();
end;

method term();
end;

method astoreScore(double inCLAGE, double inCLNO, double inDEBTINC,
                   double inDELINQ, double inNINQ, double inVALUE,
                   in_out double out_P_, in_out double outP__EVENT_0,
                   in_out double outP__EVENT_1, in_out nchar outI__EVENT_,
                   in_out nchar out_WARN_);
    CLAGE = inCLAGE;
    CLNO = inCLNO;
    DEBTINC = inDEBTINC;
    DELINQ = inDELINQ;
    NINQ = inNINQ;
    VALUE = inVALUE;

    preCode();
    sc.scoreRecord();
    postCode();

    out_P_ = _P_;
    outP__EVENT_0 = P__EVENT_0;
    outP__EVENT_1 = P__EVENT_1;
    outI__EVENT_ = I__EVENT_;
    out_WARN_ = _WARN_;
end;

endpackage;
```

Configuring ASTORE File System Paths

In order to publish decisions that use analytic store models to a SAS Micro Analytic Service destination, you must configure access to the location where the ASTORE files are located. Also, users who need to work with analytic store models must have Read and Write access to ASTORE directories. For more information, see "Configuring Access to Analytic Store Model Files" in *SAS Viya Administration: Models*.

Composite Modules

Composite modules enable DS2 code running in SAS Micro Analytic Service to call one or more analytic store models. A composite module consists of one DS2 module and zero or more analytic store models, which are known as the members of the composite. All members of a composite module are published, replaced, or deleted as a set. For example, if one member of a composite module fails to publish, then all members of the composite fail to publish. Also, any prior revisions of the composite module remain unchanged.

Like other module types, the revisions of a composite module are owned by a module context, have dictionaries, and can be queried for compilation messages, creation date-times, and so on. The members of a composite module can include either a DS2 module

or an analytic store model, or both. Other module types, such as Python and C, are not supported as composite members.

Only the methods of the DS2 module of the composite are exposed for client applications to call.

To avoid name collisions with DS2 modules that exist outside a composite module, each composite revision has its own namespace. Therefore, each composite module must be self-contained and without dependencies on any non-member module. Unlike other module types, composite modules can be called only externally (for example, driven by SAS Event Stream Processing events).

In SAS Event Stream Processing, a composite module is defined by specifying a DS2 module that calls an analytic store model or models in the **<mas-module>** tag and by including the new **<module-members>** and **<module-member>**sub-tags of **<mas-module>**. Each **<module-member>** sub-tag should specify the location of an ASTORE file on disk. The following example is an excerpt from an ESP continuous query XML definition:

```
<mas-modules>

 <mas-module language="ds2" module="module_1" func-names='astoreScore'>
  <code-file>/your_ds2_folder/your_astore_caller.ds2</code-file>
  <module-members>
   <module-member member='astore_1'
    SHAkey='EB3D1CA20AA0CB74465D25EEE2290E13692AF750' type='astore'>
    <code-file>/your_astore_model_folder/your_astore_modelcode-file>/
    your_astore_model_folder/your_astore_model>
   </module-member>
  </module-members>
 </mas-module>

</mas-modules>
```

For more information, see *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

Chapter 6

State Sharing between Modules

Overview **39**

Shared Vectors **40**
Overview 40
State Vector Types 40
Local State Vector Methods 42
Shared State Vector Methods 42
Setter and Getter Examples 44

Shared Hash Tables **46**
Overview 46
About Using Shared Hash Tables in DS2 46
Methods That Operate on the Default Shared Hash Table 48
Default Shared Hash Table Example 49
Methods That Operate on Non-default Shared Hash Tables 49

Overview

SAS Micro Analytic Service provides two ways to share data between modules that are executing within a user context: shared vectors and shared hash tables. *Shared vectors* are collections of data values. *Shared hash tables* are containers of stored vectors; the vectors accessed by using keys.

When it is possible to represent the data, or *state*, that you want to share across modules by a small number of vectors, the vectors can be shared with other modules by name. However, vector lookup by name is a linear search and is therefore inefficient when larger numbers of vectors are present. In such cases, shared hash tables are highly recommended because of their efficiency.

When using shared hash tables, an efficient non-cryptographic hashing function is applied to a key to quickly compute the desired vector's location within the hash table. Shared hash tables also use non-locking synchronization mechanisms to further increase efficiency.

Whether using shared vectors or shared hash tables, DS2 authors can use the MASState package to create, share, retrieve, and delete data.

Important: SAS Micro Analytic Service shared state vectors and shared hash tables are available only for DS2 modules. They are not supported for Python modules.

Important: These features support in-memory state sharing. They are not intended for state-sharing across cluster nodes.

Shared Vectors

Overview

Collections of state data fields that are managed as a unit are referred to as *state vectors*. Here are some key points about state vectors:

- A state vector contains one or more values, which are referred to by vector name and a zero-based index.
- The data values in a state vector can contain the same data types or a mix of data types.
- The number of data elements that is contained in a state vector is limited only by the available memory.
- A state vector is similar to a database record in that it can contain multiple data values of various types. However, it differs from a database record in that data values are positional, rather than organized in named columns.
- You can initialize a shared state vector from a DS2 package method, including the init() and constructor methods.
- A shared state vector name must be unique within the current user context. State vector values can have any of the following DS2 data types:
  - BIGINT
  - BINARY
  - DOUBLE
  - DOUBLE ARRAY
  - INTEGER
  - INTEGER ARRAY
  - VARCHAR
  - VARCHAR ARRAY

  Note: Binary data handling requires that you work within the limitations that are briefly discussed in a note in "Scalar Setters Example" on page 44. In SAS Micro Analytic Service, *binary data* typically refers to binary or character long objects. These can be expressed as pointer and length pairs or as character strings. Because DS2 does not support pointers directly, operations on binary data are typically performed with string manipulation functions.

State Vector Types

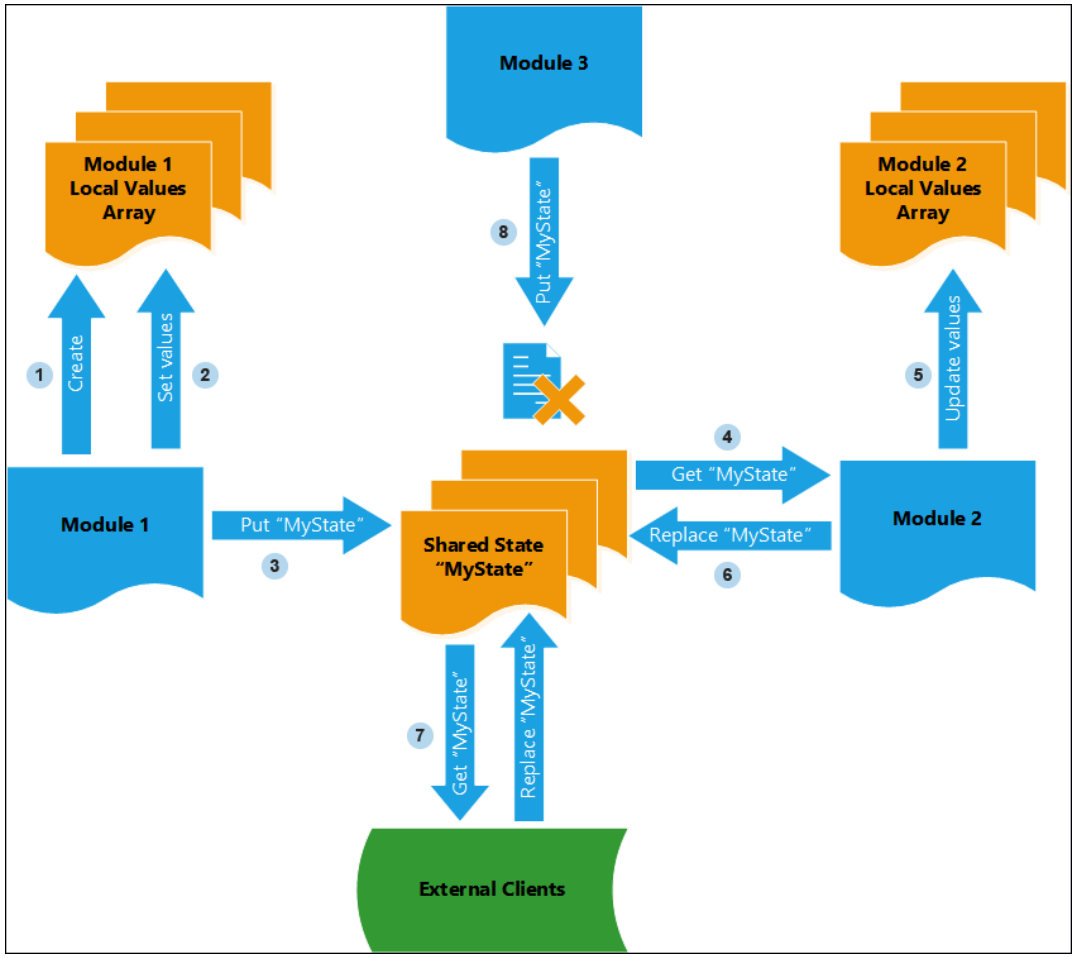
There are two categories of MASState package methods— those that operate on local state vectors and those that control state vector sharing.

Setting and retrieving individual values is always performed using local state vectors. When a shared state vector is fetched, a local copy of that vector is created and returned to the caller.

Similarly, when a state vector is shared, a copy of the local vector is created and made centrally available for fetching by other modules and transactions.

Working with local state vectors has the advantage of allowing a set of values to be updated and shared as a unit. This eliminates race conditions that could otherwise occur, and enables consistent and complete state representations.

Figure 6.1 *The State Vector Sharing Process*



1 Module 1 creates a local values array.

2 Module 1 sets the values for the array.

3 These values are published as a shared state vector and assigned the name MyState. This makes a deep copy of the vector.

4 Module 2 retrieves the MyState local vector. This makes a deep copy of the vector.

5 Module 2 updates the values and replaces the values in the local values array.

6 Module 2 replaces the values in the MyState shared state vector.

7 External clients retrieve and replace values for the MyState shared state vector.

8 Module 3 attempts to create a shared state vector called MyState. This is rejected because a shared state vector with that name already exists.

The MASState package includes 28 methods. The following sections contain usage examples for each of these methods.

Note that each example assumes that an instance of the MASState package, called st, has been created:

```
dcl package masstate st()
```

Local State Vector Methods

The following methods control the creation and deletion of local vectors.

createVector(name, size)

This method creates a local state vector with the specified name, and space for the number of values that is indicted by the specified size. The following example creates a local state vector named MyVector with a size of 4:

```
rc = st.createVector('MyVector', 4);
```

deleteVector(name)

This method deletes the local state vector referenced by name. The following example deletes the local vector created above:

```
rc = st.deleteVector('MyVector');
```

deleteAllVectors()

This method deletes all local vectors. The following example deletes all local vectors managed by the current MASState package instance:

```
rc = st.deleteAllVectors();
```

Shared State Vector Methods

The following methods control the sharing and unsharing of state vectors with other modules, and across transaction boundaries.

shareVector(name)

This method creates a copy of the named local state vector and makes it accessible to other modules within the current user context. The name passed to shareVector() must be unique within the user context. Otherwise, a duplicate name error is returned and the vector is not shared. To update an existing shared state vector, call replaceSharedVector().

```
method setValuesAndShareVector(in_out int rc);

    /* Create local vector */
    rc = st.createVector('MyVector', 4);

    /* Populate it with values*/
    rc = st.setInt('MyVector', 0, 100);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 1, 200);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 2, 300);
    if (rc ne 0) then return;
```

```
    rc = st.setInt('MyVector', 3, 400);
    if (rc ne 0) then return;

    /* Share vector with other modules */
    rc = st.shareVector('MyVector');
end;
```

fetchSharedVector(name)

This method fetches the shared state vector referenced by name and returns a local copy of it. It is used to retrieve stateful data that has been published or updated by other modules. After calling this method, the MASState package instance holds a local copy of the shared state vector, which can be referenced by name.

```
method fetchSharedVector(in_out int rc);
    rc = st.fetchSharedVector('MyVector');
end;
```

unshareVector(name)

This method removes sharing for the vector referenced by name. The shared copy of the vector is deleted from the current user context, and modules are no longer able to access it. If no shared vector with the given name exists, this is considered a valid condition and unshareVector() does not return an error. The unshareVector() method does not affect a local state vector.

```
method unshareVector(in_out int rc);
    rc = st.unshareVector('MyVector');
end;
```

replaceSharedVector(name)

This method creates a copy of the named local state vector and replaces the existing shared state vector of the same name, making the updated data accessible to other modules within the user context. The name that is passed to replaceSharedVector() must refer to an existing shared state vector. Otherwise, a **`not found`** error is returned and the data is not shared.

```
method setNewValuesAndReplaceSharedVector(in_out int rc);

    /* Populate vector */
    rc = st.setInt('MyVector', 0, 111);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 1, 222);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 2, 333);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 3, 444);
    if (rc ne 0) then return;

    /* Share vector with other modules */
    rc = st.replaceSharedVector('MyVector');
end;
```

isVectorShared(name)

This method returns integer 1 (TRUE) if a shared state vector with the given name exists within the current user context. Otherwise, it returns integer 0 (FALSE).

```
method isVectorShared(in_out int result);
```

```
    result = st.isVectorShared('MyVector');
end;
```

Setter and Getter Examples

Setter and getter methods are provided for each data type. These methods operate on local vectors only. Individual data items are referenced by local vector name and by the zero-based index of the data value.

The examples in this section illustrate each type-specific setter method. The MASState package guards against errors such as index out of range and invalid data. As a best practice, you should check return codes, and if applicable, return them to the caller.

Scalar Setters Example

```
method testScalarSetters(varchar(32) strVal,
                         int intVal,
                         bigint longVal,
                         double dblVal,
                         bigint refVal,
                         bigint refSize,
                         in_out int rc);
    rc = -1;

    /* Populate the vector with scalars of each type */
    rc = st.setString('AllScalarsVector', 0, strVal);
    if (rc ne 0) then return;
    rc = st.setInt('AllScalarsVector', 1, intVal);
    if (rc ne 0) then return;
    rc = st.setLong('AllScalarsVector', 2, longVal);
    if (rc ne 0) then return;
    rc = st.setDouble('AllScalarsVector', 3, dblVal);
    if (rc ne 0) then return;
    rc = st.setReference('AllScalarsVector', 4, refVal, refSize);
    if (rc ne 0) then return;
end;
```

Note: setReference() accepts a bigint reference value (for example, a pointer to a BLOB or other binary data in memory) and a size (BLOB size in bytes or length of other binary data). This is due to current limitations of the DS2 BINARY data type. The getReference method returns a DS2 BINARY data type. (See "Scalar Getters Example" on page 45.) The asymmetrical nature of this setter/getter pair is due to limitations with BINARY processing that exist only on the setter side. With the exception of BINARY, all other data types are handled symmetrically.

Array Setters Example

```
method testArraySetters(varchar(32) strVal[3],
                        int intVal[3],
                        bigint longVal[3],
                        double dblVal[3],
                        in_out int rc);
    rc = -1;

    /* Populate the vector with arrays of each type */
    rc = st.setStringArray('AllArraysVector', 0, strVal);
    if (rc ne 0) then return;
```

```
    rc = st.setIntArray('AllArraysVector', 1, intVal);
    if (rc ne 0) then return;
    rc = st.setLongArray('AllArraysVector', 2, longVal);
    if (rc ne 0) then return;
    rc = st.setDoubleArray('AllArraysVector', 3, dblVal);
    if (rc ne 0) then return;
end;
```

Scalar Getters Example

```
method testScalarGetters(in_out varchar strVal,
                         in_out int intVal,
                         in_out bigint longVal,
                         in_out double dblVal,
                         in_out binary refVal,
                         in_out int rc);

    /* Retrieve scalars of each type from the vector */
    strVal = st.getString('AllScalarsVector', 0);
    if (missing(strVal)) then do;
        rc = -1;
        return;
    end;
    intVal = st.getInt('AllScalarsVector', 1);
    if (missing(intVal)) then do;
        rc = -1;
        return;
    end;
    longVal = st.getLong('AllScalarsVector', 2);
    if (missing(longVal)) then do;
        rc = -1;
        return;
    end;
    dblVal = st.getDouble('AllScalarsVector', 3);
    if (missing(dblVal)) then do;
        rc = -1;
        return;
    end;
    refVal = st.getReference('AllScalarsVector', 4);
end;
```

Note that the reference value is returned as a DS2 BINARY type, as indicated in "Scalar Setters Example" on page 44.

Array Getters Example

```
method testArrayGetters(in_out varchar strVal[3],
                        in_out int intVal[3],
                        in_out bigint longVal[3],
                        in_out double dblVal[3],
                        in_out int rc);

    /* Retrieve arrays of each type from the vector */
    st.getStringArray('AllArraysVector', 0, strVal, rc);
    if (rc ne 0) then return;
    st.getIntArray('AllArraysVector', 1, intVal, rc);
    if (rc ne 0) then return;
```

```
        st.getLongArray('AllArraysVector', 2, longVal, rc);
        if (rc ne 0) then return;
        st.getDoubleArray('AllArraysVector', 3, dblVal, rc);
    end;
```

Shared Hash Tables

Overview

SAS Micro Analytic Service shared hash tables enable high-performance sharing of in-memory stateful data between modules and across transactions. Shared hash tables consist of key/value pairs, where the keys are strings and the values are state vectors. For more information about state vectors, see the previous section "Shared Vectors".

Here are some key points about shared hash tables:

- State vectors with different sizes can reside within the same shared hash table.
- Shared hash tables are visible to all modules within the same user context.
- Up to eight hash tables can exist per user context, and each hash table can contain up to 2,147,483,659 state vectors. Each state vector can contain any number of data elements.
- You can initialize a shared state vector from a DS2 package method, including the init() and constructor methods.

About Using Shared Hash Tables in DS2

The MASState package contains all the methods that are required for DS2 modules to share data across SAS Micro Analytic Service modules and transaction boundaries. These methods include operations on local state vectors and on shared hash tables.

Data can be shared among modules when you do either of the following:

- call methods that create a local state vector, populating it with values, and then putting it in a shared hash table.
- call methods that get an existing vector from a shared hash table (which makes a local copy), modifying its contents, and then replacing the vector in the hash table.

Shared hash tables are accessible by all DS2 modules within a user context.

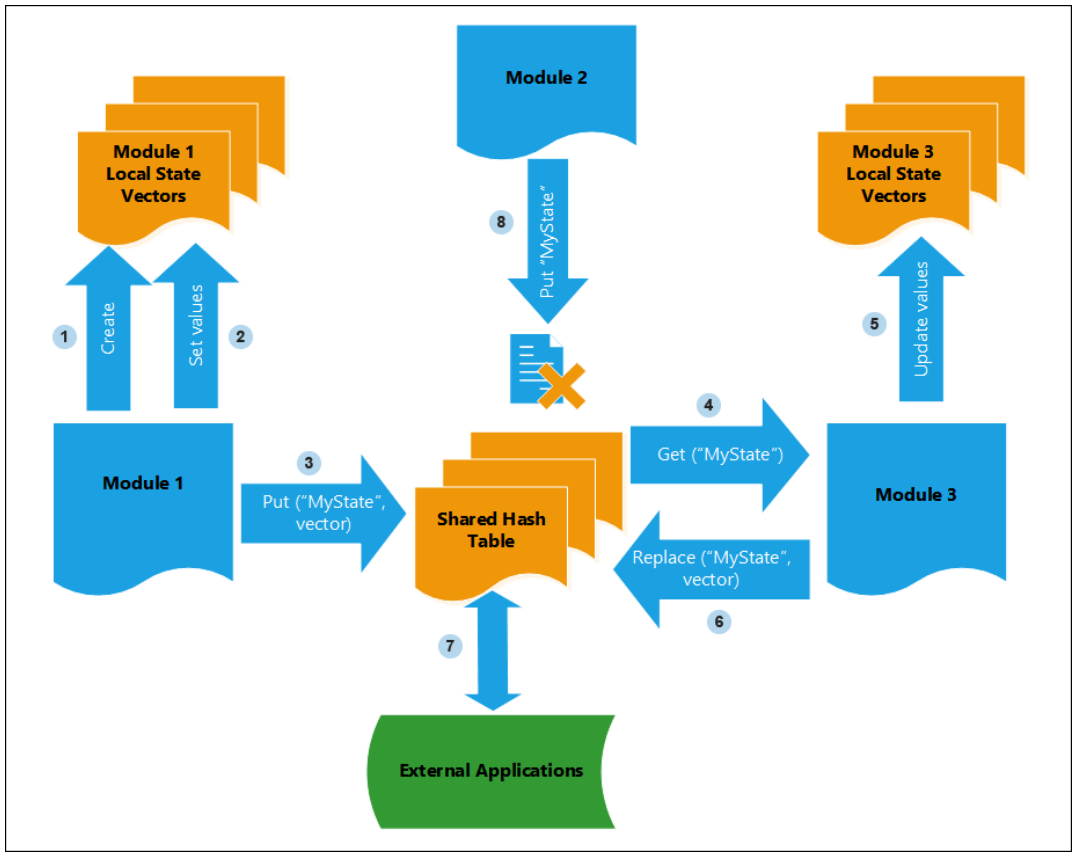
When you create a new local state vector, you assign it a name. The name must be unique within the hash table in which the vector will be stored. This name is used as follows:

- as a key when subsequently storing the vector in a shared hash table.

  That is, the name is used internally as input to a hashing algorithm that quickly computes the hash table location where the vector will be stored.
- when deleting the state vector.
- when storing or retrieving state vector data values.
- when retrieving the vector from a shared hash table.
- when replacing the vector within a shared hash table.

Up to eight shared hash tables can be defined per user context. Hash tables are referenced by index numbers zero through seven, where index zero refers to the default hash table. The default hash table is created automatically when a new user context is created. It is operated on by convenience methods that omit the table index argument. The convenience methods are clear(), isEmpty(), size(), containsKey(), put(), get(), replace(), and remove(). They are described in "Methods That Operate on the Default Shared Hash Table" on page 48.

Figure 6.2 *The Shared Hash Table Process*



1 Module 1 creates a local state vector.

2 Module 1 sets the values for the local state vector.

3 Module 1 puts these values, contained in the MyState vector, into a shared hash table.

4 Module 3 gets the MyState vector.

5 Module 3 updates the values in its local state vector.

6 Module 3 replaces the MyState state vector in the shared hash table.

7 External applications access the shared hash table to retrieve and replace the MyState state vector.

8 Module 2 attempts to store a state vector called MyState in the shared hash table. This is rejected because a state vector with that name already exists in the table.

Methods That Operate on the Default Shared Hash Table

Method Signature	Description
int clear()	Removes all state vectors from the default hash table. Returns zero if successful, and nonzero otherwise.
int isEmpty()	Returns 1 if the default hash table contains no state vectors, and zero otherwise.
bigint size()	Returns the number of state vectors currently in the default hash table.
int containsKey(key)	Returns 1 if the default hash table contains a state vector with a name matching key, and zero otherwise.
int put(key)	Inserts the state vector with the name indicated by the key into the default shared hash table. Returns zero if successful. Nonzero result codes are returned if a duplicate key exists in the default hash table or if a local state vector with a name matching key does not exist.
int get(key)	Finds a state vector in the default shared hash table with a name matching key. If found, a local copy of the state vector is made, and a zero result code is returned. If not found, a nonzero result code is returned. *Note:* If a local state vector with a name matching key already exists, and a state vector matching the key is found in the default hash table, then the existing local state vector is overwritten with the data values that are retrieved from the default shared hash table.
int replace(key)	Finds a state vector in the default shared hash table with a name matching key. If found, the state vector in the default hash table is replaced with a copy of the corresponding local state vector and a zero result code is returned. Nonzero result codes are returned if the key is not found in the default hash table, or if a local state vector with a name matching key does not exist.
int Remove(key)	Finds a state vector in the default shared hash table with a name matching key. If found, removes it and returns a zero result code. A nonzero result code is returned if the key does not exist in the default hash table.

Default Shared Hash Table Example

In the following example, method createAndPutVector() inserts a new state vector containing two integer values into the default shared hash table. Method incrementSharedValue() retrieves a state vector, named MyVector, from the default shared hash table, making a local copy. It increments the integer data value within the vector and then replaces the MyVector state vector in the default shared hash table.

```
ds2_options sas;
package statepkgtest/overwrite=yes;
dcl package masstate st();

 method createAndPutVector(varchar(32) key, in_out int rc);
   rc = st.createVector(key, 2);
   rc = st.setInt(key, 0, 100);
   if (rc ne 0) then return;
   rc = st.setInt(key, 1, 200);
   if (rc ne 0) then return;
   rc = st.put(key);
   rc = st.deleteVector(key);
end;
method incrementSharedValue(in_out int rc, in_out int int0Val);
   rc = st.get('MyVector');
   if (rc eq 0) then do;
       int0Val = st.getInt('MyVector', 0);
       int0Val = int0Val + 1;
       rc = st.setInt('MyVector', 0, int0Val);
       rc = st.replace('MyVector');
   end;
end;
endpackage;
```

Methods That Operate on Non-default Shared Hash Tables

Note: For the methods in the table, the following arguments apply:

- **`tableIndex`** indicates the hash table (0-7) on which to operate.
- **`key`** is a string value that uniquely identifies a vector within the hash table.

Method Signature	Description
int hashTblCreate(**`tableIndex`**)	Creates a new empty hash table, which can be referenced by the given table index. Returns zero if successful, and nonzero otherwise.
int hashTblDestroy(**`tableIndex`**)	Removes all state vectors from the indicated hash table, and then deletes the table. Returns zero if successful, and nonzero otherwise.
int hashTblClear(**`tableIndex`**)	Removes all state vectors from the indicated hash table. Returns zero if successful, and nonzero otherwise.

Method Signature	Description
int hashTblIsEmpty(**tableIndex**)	Returns 1 if the indicated hash table contains no state vectors, and zero otherwise.
bigint hashTblSize(**tableIndex**)	Returns the number of state vectors currently in the indicated hash table.
int hashTblContainsKey(**tableIndex**, **key**)	Returns 1 if the indicated hash table contains a state vector with a name matching key, and zero otherwise.
int hashTblPut(**tableIndex**, **key**)	Inserts the state vector into the indicated hash table at the position indicated by key. Returns zero if successful. Nonzero result codes are returned if a duplicate key already exists in the indicated hash table, or if a local state vector with a name matching key does not exist.
int hashTblGet(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key. If found, a local copy of the state vector is made, and a zero result code is returned. If not found, a nonzero result code is returned. *Note:* If a local state vector with a name matching key already exists, and a state vector matching the key is found in the indicated hash table, then the existing local state vector is overwritten with the data values that are retrieved from the hash table.
int hashTblReplace(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key. If found, the state vector in the indicated hash table is replaced with a copy of the corresponding local state vector and a zero result code is returned. Nonzero result codes are returned if the key is not found in the hash table, or if a local state vector with a name matching key does not exist.
int hashTblRemove(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key and, if found, removes it and returns a zero result code. A nonzero result code is returned if the key does not exist in the hash table.

Chapter 7

Best Practices for DS2 Programming in SAS Event Stream Processing

Overview 51
Return Results 51
Global Packages versus Local Packages 52
Overview 52
Example of Optimized Code 52
Example of Poorly Optimized Code 52
Replacing SCAN (and TRANWRD) with DS2 Code 53
Hash Package 55
Character-to-Numeric Conversions 55
Data Type Conversions 55
Passing Character Values to Methods 56
Performing the Computation Once 56
Moving Invariant Computations Out of Loops 56

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.

Return Results

If a DS2 method, or any method it calls, can result in a status code or failure, always include a method output argument for returning the result to the caller.

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 the CPU load due to 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 */
```

Data Type Conversions

When a source or derived event window includes an array of a particular type, and the corresponding argument type in the module method is another type, a data type

conversion can occur automatically. This happens only under certain circumstances. For more information, see "Implicit Data Type Conversions" on page 32.

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 8

Python Support in SAS Micro Analytic Service

Introduction **59**

Example **60**

Public and Private Methods **62**
Overview 62
About Private Methods 62
About Public Methods 62
Return Values 63
Examples: Public and Private Methods 64

Working with Python and SAS Micro Analytic Service **65**

Configuring Python for SAS Event Stream Processing **66**
Environment Configuration 66

Introduction

SAS Micro Analytic Service supports modules that are written in the Python programming language. A Python module represents a group of related Python functions.

Input arguments are given in the function's argument list. The objects, variables, and expressions listed in a Python function's return statement are positional with respect to the output variables.

The output variables are listed in the function’s "Output:" docstring that is specified in the first statement of the function. Any method that includes the "Output:" docstring is considered a public method. Otherwise, it is considered a private method. For information, see the sections later in this chapter.

Input and output argument names live in a single namespace and therefore cannot be the same. This means that update arguments are not supported. This is true for all module types in SAS Micro Analytic Service, even though the Python language does not enforce such a restriction.

Here is an example of a Python public function that can be hosted by SAS Micro Analytic Service.

```
import sys
import math
import pandas as pd
```

```
import numpy as np
def nppd(a):
    "Output: ser1"
    npa = np.array([[1,2,3],[4,5,6]])
    ser1 = pd.Series([212, a, -273])
    return ser1.tolist()

def trucks(Eng_Load, Oil_Temp, Eng_RPM):
    "Output: ser1, x, syspath"
    inputs = pd.Series([Eng_Load, Oil_Temp, Eng_RPM])
    b = np.arange(100)
    number = 0
    for index, item in enumerate(inputs):
        number += item + b[index + 7]
    # is it even or odd?
    x = math.fmod(number, 2)
    return nppd(Oil_Temp), x, getsyspath()

def getsyspath():
    "Output: p"
    p = [None] * 50
    # print(sys.path)
    syspaths = sys.path
    i = 0
    for path in syspaths:
       p[i] = path
       i = i + 1
    return p
```

Here is an example of a Python public function that has input arguments a and b, and no output.

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

After Python is configured, see Appendix 1, “Executing Python Modules in DS2 Modules,” on page 139 for additional information.

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()

    if inBool == None:
        outBool = None
    else:
        # Reverse value of the Boolean.
        outBool = not inBool

    if inLong == None:
        outLong = None
    else:
        # Add 10 to long.
        outLong = inLong + 10

    if inDouble == None:
        outDouble = None
    else:
        # Add 10.1 to the double.
        outDouble = inDouble + 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

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)

if inMoney == None:
    outMoney = None
else:
    # Add 25 cents.
    outMoney = inMoney + 0.25


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

Public and Private Methods

Overview

SAS Micro Analytic Service enables the use of hosting public and private methods, where a method is a Python function. Note that public and private methods are SAS Micro Analytic Service concepts, and are not Python features.

In general, any method that includes the "Output:" docstring is considered a public method. If a method does not have the "Output:" docstring, then it is considered a private method. However, there are syntax requirements that must be followed for the docstring and the output arguments. For details, see “About Public Methods” on page 62.

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.

About Private Methods

Here are details about using a private method:

- A private method can be called internally by other methods (either public or private).
- A private method cannot be called directly (externally).
- Private methods are useful when used as utility functions within a package.

About Public Methods

Here are details about using a public method:

- For a function that has at least one output argument, there must be a space between "Output:" and the first output argument name. For examples, see the next section "Examples: Public and Private Methods" on page 64.
- When there is more than one output argument, the output argument names must be comma separated.
- Line two of the function must begin with a docstring, and the first non-whitespace token must be "Output:".
- All public functions that return 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:" docstring to indicate a public function. It should simply be "Output:", with no output arguments listed.
- Order does matter. Therefore, the order in the return statement must match the order in the "Output:" line. A best practice is to copy and paste from one to the other.

Return Values

When Python integers are returned, they are converted to int64_t. If the Python integer is too large for the int64_t type, a run-time error is reported.

Although Python packages might support abstract types that provide an additional layer on top of the built-in scalar types, such as integer and float, they cannot be returned from SAS Micro Analytic Service public functions. Packages that have these abstract types provide functions that can be used to extract the built-in type. Here is an example:

```
return numpy.float(x), numpy.int(y)
```

Note the following information about the return statement for Python functions:

- Functions that return nothing do not require a return statement. Python returns the NoneType object. Therefore, an empty return statement (**`return`**) and returning None (**`return None`**) are equivalent.
- Functions that return a single argument can return None. SAS Micro Analytic Service maps the None value to the specific Missing value for the expected static type.
- Functions that return multiple values can be formatted with or without parentheses (for example, **`return a,b,c`** or **`return (a,b,c)`**).

  Note: An attempt to return None from a function that expects multiple values is invalid.

When return values are transferred between Python and SAS Micro Analytic Service, the following rules are enforced by SAS Micro Analytic Service:

- A singleton value (non-tuple) is mapped to a tuple with a single value.
- Tuples map to multiple return arguments.
- Arrays and tuples cannot be used interchangeably. The Python tuple type is used to return multiple values. The Python list type is used for SAS Micro Analytic Service arrays.
- The argument count must match.

The following code sample and table show some examples that illustrate the application of these rules.

```
def one(a):
"output: b"
return oneVal

def two(a,b)
"output: a,b"
return twoVal
```

Python Value Assignment	SAS Micro Analytic Service Value Assignment
`oneVal = 1`	`oneVal = (1)` (one integer value)
`oneVal = 1,`	`oneVal = (1)` (one integer value)
`oneVal = (1,)`	`oneVal = (1)` (one integer value)
`oneVal = (1, 2)`	An error occurs. It indicates that the expected output variables do not match the actual output variables.
`twoVal = 1,2`	`twoVal = (1,2)` (two integer values)
`twoVal = (1,2),`	An error occurs. It indicates that an unknown tuple type is present.
`twoVal = ([1],[2])`	`twoVal = ([1],[2])` (two integer arrays)

Examples: Public and Private Methods

As mentioned previously in this chapter, for a method to be public, the output variables must be listed in the function's "Output:" docstring that is specified on the first statement of the function. This is the second line of the method, immediately following the "def" line.

Here are some examples. Note that the fun2 function would be considered private because the docstring does not begin with "Output:".

```
def fun1( a, b ):
    "Output:"
    """ This is a public function,
        but has no output args."""
    return

def fun2( a, b ):
    """This will be private since the docstring doesn't begin with Output:
```

```
      Output: x, y, z"""
    return a+2, b*4, a/b

def fun3( a, b ):
    '     Output: x, y, z'
    ''' multi
        Line
        doc string'''
    return a+2, b*4, a/b

def fun4( a, b ):
    '''  Output: x, y, z'''
    """ multi
        Line
        doc string"""
    return a+2, b*4, a/b

def fun5( a, b ):
    '''Output: q, r, s,
               t, u,
               v,
               w'''
    return a+2, b*4, a/b
```

Working with Python and SAS Micro Analytic Service

When you work with Python and SAS Micro Analytic Service, note the following:

- Multi-tenant deployments are not supported.
- When sharing modules across other modules, because all modules belong to the same tenant, you should understand the following logic:

  If module A and module B each import module C, updates to module C affect both module A and module B.
- All client requests are passed to SAS Micro Analytic Service with an OAuth access token. This token is not passed to the Python subprocess. Therefore, Python code cannot connect to a SAS service that requires an access token.

  Note: This is a SAS Micro Analytic Service limitation on Python modules.
- SAS Micro Analytic Service Python interface usage is prohibited when any of the following situations exist:
  - The application enabled MASHostAccessDeny when initializing SAS Micro Analytic Service.
  - The MASHostAccess=DENY environment variable is set.
  - The application provides MASHostOSID credentials that fail to authenticate.
  - The CAS session is not running under a host operating system identity and it attempts to run an action that uses SAS Micro Analytic Service.
- In this release of SAS Micro Analytic Service, maintaining multiple revisions of a Python module is not supported. A subsequent publish request on the same Python

module context replaces the existing revision. This means that the revision number always remains at 1.

For example, consider the following two Python modules:

test.py:

```
def execute(a,b):
   "Output: c"
   c = (a+b)
   return c

def score(x):
   "Output: y"
   return x * 0.5;
```

test2.py:

```
def execute(a,b,c):
   "Output: d"
   d = (a+b+c)
   return d;
```

If you publish test.py and then publish test2.py to the same module context, the first execute function is replaced by the second one. The new execute function has an additional input argument (c) and a different output argument (d). Also, the new revision no longer includes the score function.

Configuring Python for SAS Event Stream Processing

Environment Configuration

SAS Micro Analytic Service does not require a specific version of Python. However, it is possible that Python code that you publish to SAS Micro Analytic Service might have dependencies to a specific version of Python or Python packages. To configure Python for use within SAS Micro Analytic Service, note the following requirements:

- The MAS_PYPATH environment variable must be set. It specifies the absolute path to the Python executable file. Here are some examples:
  - UNIX platform:

    ```
    MAS_PYPATH=/usr/bin/python
    export MAS_PYPATH
    ```

  - Windows platform:

    ```
    set MAS_PYPATH=c:\python\python.exe
    ```

- When SAS Micro Analytic Service launches the Python client, it waits for the environment to reconnect. By default, this wait time is 30 seconds. To change the wait time, you can set the MAS_PYWAIT environment variable and assign the applicable value (in milliseconds).
- Note the following important information about using Python:
  - 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 -*-
```

- 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"
  ```

- Python 3.*x* uses UTF-8 as the default encoding. Therefore, the encoding issues noted above 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.

- For Python environments prior to version 3.7, ensure that the LANG environment variable is set as follows:

  ```
  LANG=*.UTF-8
  ```

  The asterisk (*) corresponds to the applicable locale value for your environment. Some examples are *en_US.UTF-8* (English United States), *fr-FR.UTF-8* (French France), or *de-CH.UTF-8* (German Switzerland).

  Note: If the LANG environment variable is not defined, SAS Micro Analytic Service sets its value to **`en_US.UTF-8`**.

- Note the following information when you set environment variables (such as PYTHONPATH) in the sub-processes that are launched to run on Python:

  - In a UNIX environment, exported shell environment variables are inherited by the sub-processes.

  - In a Windows environment, you must define system and user environment variables in **Control Panel** ⇨ **System and Security** ⇨ **System** ⇨ **Advanced system settings** ⇨ **Environment Variables**.

Chapter 9

SAS Micro Analytic Service Logging and Deployment

SAS Micro Analytic Service Logging **69**
Overview 69
Loggers 69
Deployment **70**

SAS Micro Analytic Service Logging

Overview

SAS Micro Analytic Service uses the SAS Logging Facility. For more information, see *SAS Viya Administration: Logging*. 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, access the SAS product documentation at http://support.sas.com.

Compiler messages are retrieved and logged by SAS Event Stream Processing when it publishes a module to SAS Micro Analytic Service.

Loggers

Logger names that start with the following prefixes apply to the SAS Micro Analytic Service core: Admin, App, Audit, Perf.

Here are some important SAS Micro Analytic Service core loggers:

Name	Events Logged	Default Value
App.tk.MAS	Method execution events	INFO
App.tk.MAS.Service	Start-up and shutdown events and the SAS Micro Analytic Service version number	INFO
App.tk.MAS.Python	Python events	INFO

Name	Events Logged	Default Value
App.tk.MAS.CodeGen	Compilation messages produced during an attempt to publish. When a publish request fails, error information is logged regardless of the App.tk.MAS.CodeGen logger level.	FATAL
App.SQLServices.license App.license DataFlux.Licensing	Licensing events	ERROR
Audit.Table.Connection	Database connection events	INFO

Loggers that start with App.TableServices.DS2.Runtime. or App.TableServices.DS2.Config. can be used to diagnose DS2 problems. Module code might also use other loggers.

When diagnosing DS2 problems, it is important to note that the App.TableServices.DS2.Runtime.* and App.TableServices.DS2.Config.* loggers do not inherit configuration from their ancestors. They must be configured explicitly, if you want to capture logging events that are directed to those loggers. It is recommended that you configure them only when diagnosing a DS2 problem since the additional logging traffic affects performance. For more information about those DS2 loggers, see the "DS2 Loggers" section of *SAS DS2 Programmer's Guide*.

Detailed information about operations such as compilation start and finish is logged at the DEBUG level. Warning and error conditions are logged at the WARN or ERROR levels, as appropriate.

Note: When App.tk.MAS.Service logger's level is set to DEBUG or TRACE, you see a message logging event that provides the SAS Micro Analytic Service version number in the log. Here is an example:

```
May14 15:12:49 [00000007] DEBUG App.tk.MAS.Service -
Micro Analytic Service version information: 5.3, V.03.04M0P05052019,
Mon May  6 03:00:21 EDT 2019
```

Important: If you change the value for any SAS Micro Analytic Service core loggers, you must restart SAS Micro Analytic Service for the change to take effect.

Deployment

SAS Micro Analytic Service is deployed automatically when SAS Event Stream Processing is deployed. If necessary, you can add environment customizations to your .bashrc file. For example, if you will use Python modules, you must complete the deployment and configuration steps that are described in "Configuring Python for SAS Event Stream Processing" on page 66.

If you encounter stack overflow errors when you are using the default SAS Micro Analytic Service worker thread stack size of 8 MB, you can override the default value by adding a definition of the MASTktstacksize environment variable to your .bashrc file. Note the following information when assigning a value to MASTktstacksize:

- You can specify a value using a unit of kilobytes or megabytes. The default value unit is kilobytes.
- To specify a value using megabytes, you must include “m” or “M” as a suffix following the number (for example, 10M or 10m).
- To specify a value using kilobytes, you can specify no suffix or add either a “k” or a “K” following the number. This means that any of the following are valid values: 10240, 10240k, 10240K.

Here are examples of different ways to assign a value of 10 MB to the MAStktstacksize environment variable:

- export MAStktstacksize=10240
- export MAStktstacksize=10240k
- export MAStktstacksize=10240K
- export MAStktstacksize=10m
- export MAStktstacksize=10M

Part 3

Using SAS Micro Analytic Service with SAS Intelligent Decisioning or SAS Model Manager

Chapter 10
DS2 Programming for SAS Micro Analytic Service *75*

Chapter 11
State Sharing between Modules *93*

Chapter 12
Best Practices for DS2 Programming in SAS Intelligent Decisioning *105*

Chapter 13
Python Support in SAS Micro Analytic Service *111*

Chapter 14
Administration *121*

Chapter 10

DS2 Programming for SAS Micro Analytic Service

Overview **76**

DS2 Source Code Prerequisites **76**

DS2 Identifiers **76**

SAS Micro Analytic Service and SAS Foundation **77**

Programming Blocks **78**

Restrictions When Working with DS2 and SAS Micro Analytic Service **78**
Character Restrictions 78
User-Defined Formats 79

Public and Private Modules and Methods **79**
Overview 79
Public Method Rules 80
Public Method Example 80
Private Method Example 82
Method Overloading 82

Argument Types Supported in Public Methods **82**
Overview 82
Supported DS2 Data Types 82
Unsupported DS2 Data Types 82
Data Grid Support 83

Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service . . . **84**

Performing Calls between SAS Micro Analytic Service Modules **84**
Overview 84
MASCall Methods 84
Examples 85

Managing Large DS2 Modules **86**
Overview 86
Asynchronous Module Creation and Update 87
SAS Micro Analytic Service Time-Out Values 87
Loading Modules On-Demand 87
Verifying Minimum Memory 88
Loading Modules from the Repository at Start-Up 88
Understanding Eventual Consistency and Module Availability 88

Composite Modules **89**

Referencing Modules and Composite Submodules **89**

Using Analytic Store Models **89**
About Analytic Store Models 89
Publishing an Analytic Store Model 90
Calling Analytic Store Models Using DS2 90
Example 90
Configuring ASTORE File System Paths 91

Overview

SAS Micro Analytic Service 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 statement 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. DS2 source code 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.

Note: DS2 supports only a specific style of comment. Comments start with the characters /*, and they end with the characters */. All characters between the starting and ending characters are part of the comment text. Comments can be nested. When there is ambiguity in determining a token, the compiler always chooses the longest possible sequence of characters that can make up a token.

DS2 Identifiers

For DS2 method, package, and argument names, SAS Micro Analytic Service supports regular identifiers and delimited identifiers. When using a delimited identifier, any character is allowed, including multi-byte and non-ASCII characters. You must begin and end delimited identifiers with double quotation marks. For complete information, see “DS2 Identifiers” in *SAS DS2 Programmer’s Guide*.

In the SAS Micro Analytic Service REST service, the names of the output parameters of a step are the same as the in_out parameters names of the method. The parameter names of the method can use delimited identifiers. However, they are difficult to use in a JSON payload that uses double quotation marks to specify field names. Therefore, the double quotation marks around the delimited identifiers for output parameters are not included in the StepOutput.

Consider the following example:

A DS2 method returns a parameter called CustomerAge that has a value of 45. The JSON output would appear as follows:

```
{
        "name"    : "CustomerAge"
        "value"    : 45
}
```

However, if an embedded space was included for readability purposes, the delimited identifier "Customer Age" must be used. Because the double quotation mark (") is a reserved character in JSON, it is represented by the escape characters **\"**. This would result in the following JSON representation:

```
{
        "name"    :    "\"Customer Age\"",
        "value"    :    45
```

Because this is difficult to read, the SAS Micro Analytic Service REST service removes the escape sequence characters. This results in the following JSON output:

```
{
        "name"    :    "Customer Age",
        "value"    "    "45
```

This is easy to read and is compatible with standard JSON tools.

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.

Here is an example PROC DS2 step that illustrates the above mechanism:

```
proc ds2;

ds2_options sas;
package myPackage/overwrite=yes;
method copyArray(char(12) in_array[4], in_out char(12) out_array[4]);
    out_array := in_array;
end;
endpackage;
run;

table _null_;
```

```
method init();
   dcl package myPackage p();
   dcl char(12) inarr[4];
   dcl char(12) outarr[4];
   inarr[1] = 'one';
   inarr[2] = 'two';
   inarr[3] = 'three';

   p.copyArray(inarr, outarr);
   put outarr[1]=;
   put outarr[2]=;
   put outarr[3]=;
end;
run;

quit;
```

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

Restrictions When Working with DS2 and SAS Micro Analytic Service

Character Restrictions

The following characters are not allowed in module IDs, package names, public method names, and submodule names:

- backslash (\)
- forward slash (/)
- period (.)
- semicolon (;)

User-Defined Formats

SAS Micro Analytic Service does not support the use of SAS sessions. Because a SAS session is required to create and access user-defined formats, SAS Micro Analytic Service does not support the use of user-defined formats.

Public and Private Modules and Methods

Overview

Public and private modules and methods are SAS Micro Analytic Service concepts, rather than DS2 features. SAS Micro Analytic Service can host modules with methods with public and private levels of visibility.

Public methods are methods of a module that are available for execution using the REST interface. In order to be available, a method can contain only those parameters that map to the following types and return a void result:

- bigint
- bigintArray
- datagrid
- dateTime
- dateTimeArray
- decimal
- decimalArray
- integer
- integerArray
- string
- stringArray

For information about the DS2 types that correspond to the above types, see "Argument Types Supported in Public Methods" on page 82.

If a method contains other types of parameters or returns a non-void value, it is considered private and cannot be executed using the REST service. However, these methods can be called by other methods.

Public modules are modules that contain public methods that can be executed using the REST interface. Module creators can choose to specify the visibility of a module. However, a public module with no public methods is effectively a private module.

Private modules and methods can be used to provide a clean interface to code that hides internal implementation. Private methods are often used as utility or library methods to help solve larger problems.

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 82.

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. The DS2 programming language does support method overloading for programs running in other environments, but not when running in SAS Micro Analytic Service.

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 included 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)

Data Grid Support

In addition to the above supported DS2 data types, if SAS Intelligent Decisioning is installed, SAS Micro Analytic Service also supports tabular data in data grid format. This format is similar to a database table but it is typically much smaller in size. It is used to hold structured data that cannot be well represented by arrays.

Public methods that contain parameters of the type DS2 package dcm_datagrid are represented in the JSON payload of the media types application/vnd.sas.microanalytic.module.step.input+json (input) and application/vnd.sas.microanalytic.module.step.output+json (output).

The JSON structure is as follows:

- The first part is called metadata. It specifies each column in the table by name and type. The assigned type can be Boolean, decimal, integer, string, or dateTime.
- The next part contains rows of data that correspond to the order and types of the columns specified in the metadata part.

Here are a few examples of how the data grid format is represented:

```
"aTable" : [{
        "metadata" : [
              { "aStringColumn" : "string" },
              { "anIntColumn" : "integer" },
        { "aFloatColumn" : "decimal" },
        { "aBooleanColumn" : "boolean" },
        { "aTimestampColumn" : "dateTime" }]
    }, {
    "data" : [[ "one", 1, 1.11, true, "2001-01-01T01:01:01.000Z" ],
              [ "two", 2, 2.22, false, "2002-02-02T02:02:02.000Z" ],
              [ "three", 3, 3.33, true, "2003-03-03T03:03:03.000Z" ],
              [ "four", 4, 4.44, false, "2004-04-04T04:04:04.000Z" ],
              [ "five", 5, 5.55, true, "2005-05-05T05:05:05.000Z" ],
        [ null, null, null, null, null ]]
    }],
    "aTableWithNoData" : [{
        "metadata" : [
              { "aStringColumn" : "string" },
              { "anIntColumn" : "integer" },
        { "aFloatColumn" : "decimal" },
        { "aBooleanColumn" : "boolean" },
        { "aTimestampColumn" : "dateTime" }]
    }, {
```

```
        "data" : []
    }],
    "aNullValuedTable" : null
```

Determining Whether DS2 Code Is Executing in SAS Micro Analytic Service

The DS2 function inmas() discovers whether SAS Micro Analytic Service is running in the current process and, if so, determines whether the current thread is a member of the SAS Micro Analytic Service worker thread pool. If it is, then the DS2 code is running inside SAS Micro Analytic Service.

The function returns 1 (TRUE) if the DS2 code is executing in SAS Micro Analytic Service, and 0 (FALSE) otherwise.

This can be useful to know when, for example, you have DS2 code that works in various locations, but not in SAS Micro Analytic Service.

Performing Calls between SAS Micro Analytic Service Modules

Overview

When a DS2 module references another DS2 package, the DS2 compiler does the following:

1. Copies, from the source code repository, the source code of the referenced package into the source code of the module to be published. The code is copied inline.

   Note: The current DS2 language infrastructure supports source code repositories only.

2. Compiles the combined source code into a single executable code stream.

A drawback to this approach is that the DS2 package code is repeated in every module that references it. This results in increased memory usage for every redundant copy of a package and longer compilation times.

SAS Micro Analytic Service resolves these issues via the DS2 MASCall package. This package contains methods that enable separate DS2 modules, published to SAS Micro Analytic Service, to call one another across separate module executables.

In addition to a smaller memory footprint and shorter compilation times, this functionality enables a library of modules to be reused by many higher-level modules without penalty.

MASCall Methods

Here are the package MASCall methods:

- allocParms(module_name, method_name)

  This method creates a parameter list for the latest revision of the specified method.

- allocRevParms(module_name, revision_number, method_name)

  This method is similar to allocParms, but it enables you to specify a revision number.

Important: The called module name is case-sensitive on UNIX systems. It is not case-sensitive on Windows systems.

After allocParms() or allocRevParms() is called, the parameter setter methods operate on the newly created parameter list.

Note that for both the setter and getter methods, **`arg_index`** is zero-based. The argument takes a numeric index value, not a method parameter name.

Here are the setter methods:

- Scalar argument setters are of the form:

  return_code = set<type>(arg_index, value)

- Array argument setters are of the form:

  return_code = set<type>Array(arg_index, array_value)

After the input values are set, you can execute the target method by making the following call, which calls the external module method that was previously specified by allocParms() or allocRevParms():

callModule()

Note: Because the method to execute was previously specified, callModule() has no parameters.

Output values returned by the method execution can be retrieved using the MASCall getter methods, which are as follows:

- Scalar argument getters are of the form:

  value = get<type>(arg_index)

- Array argument getters are of the following form.

  get<type>Array(arg_index, array_value, rc)

  Note: DS2 passes arrays and output values by reference.

After the output values are retrieved, call releaseParms() to release the parameters and other resources used for the method. This releases the memory resources used for memory execution and prevents memory leaks.

Examples

Here are DS2 MASCall package example method calls, where **`mc`** is the DS2 MASCall instance variable that is created by calling the package constructor, which takes no arguments. Here are two ways to construct a MASCall package instance named **`mc`**:

- Example 1:

```
declare package mascall mc();
```

- Example 2:

```
declare package mascall mc;
mc = _new_ mascall();
```

General example:

```
rc = mc.allocParms( module_name, method_name );
rc = mc.setDouble( 0, additionalDiscount );
```

```
rc = mc.setString( 1, currentPhone );
rc = mc.setString( 2, currentPlan );
rc = mc.callModule();
treatments = mc.getString( 3 );
mc.releaseParms();
```

The complete set of DS2 package methods follows, where **rc** is the integer return code, and **mc** is the package instance. Note that **arg_index** is zero-based. The argument takes a numeric index value, not a method parameter name.

Parameter resource management:

```
rc = mc.allocParms(    module_name, method_name );
rc = mc.allocRevParms( module_name, revision, method_name );
     mc.releaseParms();
```

Scalar argument setters:

```
rc = mc.setString(   arg_index, value );
rc = mc.setLong(     arg_index, value );
rc = mc.setInt(      arg_index, value );
rc = mc.setDouble(   arg_index, value );
```

Array argument setters:

```
rc = mc.setStringArray(   arg_index, string_array    );
rc = mc.setLongArray(     arg_index, bigint_array    );
rc = mc.setIntArray(      arg_index, integer_array   );
rc = mc.setDoubleArray(   arg_index, double_array    );
```

Execute or call a module's method:

```
rc = mc.callModule( );
```

Scalar argument getters:

```
string_var =    mc.getString(   arg_index );
long_var =      mc.getLong(     arg_index );
int_var =       mc.getInt(      arg_index );
double_var =    mc.getDouble(   arg_index );
```

Array argument getters:

```
mc.getStringArray(   arg_index, string_array,    rc );
mc.getLongArray(     arg_index, bigint_array,    rc );
mc.getIntArray(      arg_index, integer_array,   rc );
mc.getDoubleArray(   arg_index, double_array,    rc );
```

Managing Large DS2 Modules

Overview

SAS Micro Analytic Service compiles DS2 code and retains that compiled code in memory (known as the *repository*). Thus, SAS Micro Analytic Service can execute code and return responses in near real-time. A large DS2 module requires a lot of time to compile. Subsequently, this affects the time that it takes to load the module from the repository which, in turn, affects server start-up time.

Large modules also require more memory to host in a server. Memory usage is approximately proportional to the number of core threads. Determining the number of

core threads to assign is an exercise in balancing the memory and the throughput requirements of the system.

SAS Micro Analytic Service includes several features to support and manage large DS2 modules. These features enable a system administrator to balance the size and complexity of the modules against system memory, system responsiveness, and availability. This balancing process includes asynchronous calls to create or update modules, as well as time-outs on operations that create or update modules. It can also include disallowing a module to the persistence store if it takes too long to compile or there is insufficient available memory.

The topics in this section describe all these features.

Asynchronous Module Creation and Update

Calls to the SAS Micro Analytic Service REST service are governed by the time-out value that is configured for the SAS Viya web server. If the time to compile the content exceeds that value, a call times out for the client.

The SAS Micro Analytic Service jobs endpoint allows the caller to submit a Create request or an Update request that is longer than the SAS Viya web server time-out value. Some benefits include the following:

- The caller can check the status of a job and monitor its progress.
- The caller can delete jobs before they are executed or after completion.
- Outdated jobs are automatically purged. By default, this occurs after 24 hours.

For complete details about the jobs API, see the REST API documentation.

SAS Micro Analytic Service Time-Out Values

Compiling DS2 Modules

SAS Micro Analytic Service imposes a time limit for compiling DS2 modules. If the compilation time exceeds this time-out value, the module is not compiled and therefore is not persisted in the repository. Keep in mind that because extremely large modules take a long time to compile, they might not be suitable for deployment in SAS Micro Analytic Service.

This compilation time-out value is configurable. You can modify it by using the service.timeouts.maxmodulecompiletimemillis property in SAS Environment Manager.

For more information, see "supplementalProperties" on page 129.

Creating and Updating Synchronous Modules

In addition to the previously mentioned asynchronous methods, you can also create or update modules synchronously. To configure a time-out value for these methods, use the service.timeouts.maxloadwaitnonexecmillis property in SAS Environment Manager.

For more information, see "supplementalProperties" on page 129.

Loading Modules On-Demand

If a call to access a module is made, and it has not been loaded from the repository, SAS Micro Analytic Service attempts to compile the module. This might occur following a

restart if the server is still loading modules from the library or if the module was created or updated on another cluster node.

This also might occur in cases in which a module is compiled because of a query request for detailed information that can be retrieved only from a compiled module, or a request to execute that module.

Both of these situations have a corresponding configurable time-out value that you can modify by using the following configuration items:

- **query operation time-out**: service.timeouts.maxloadwaitnonexecmillis
- **execution operation time-out**: service.timeouts.maxloadwaitexecmillis

For more information, see "supplementalProperties" on page 129.

Verifying Minimum Memory

If a minimum amount of memory is not available on the server, SAS Micro Analytic Service does not compile a module. In this case, an internal error is returned to the caller. If the low memory condition occurs during the loading of the module from the repository, the messages are logged and the module is not loaded.

The logged message contains the details of available and used memory.

You can modify the minimum value limit by using the core.minfreememoryfloor configuration item.

For more information, see "supplementalProperties" on page 129.

Loading Modules from the Repository at Start-Up

When a cluster node starts, it attempts to load all the modules that are in the repository. These modules are already in the system, so loading them is prioritized over processing any job requests to create or update modules.

Because these modules were validated before they were added to the repository, errors are not expected when they load. In the rare case that a specific module fails to load, possibly because of an environmental factor (for example, a memory issue), that module is not loaded. There are no subsequent attempts to load the module unless the node is restarted.

Understanding Eventual Consistency and Module Availability

Some special considerations apply to clustered deployments with regard to large modules.

In a clustered multi-node system, modules must be compiled on all nodes of the cluster that host SAS Micro Analytic Service. The first node that receives a Create or Update request compiles the module and responds to the caller with any compilation errors. If there are no errors, SAS Micro Analytic Service saves the module code to the repository and loads it to the other nodes for compilation.

Depending on any other modules that are in the process of compiling, some nodes might not be ready to load this module. During this time, a request to execute that module succeeds only if it is serviced by a node in which the module is compiled and loaded. Modules that were previously loaded are available and requests to execute are serviced.

If a module is compiled synchronously (by using the create module or update module API) control returns to the caller as soon as the module is compiled on one node of the

cluster. Similarly, if a module is compiled asynchronously, a job status of **`completed`** is returned as soon as the module is compiled on one node of the cluster. The other nodes continue to compile the module. Eventually, all the nodes compile and load the module. However, before that happens, requests to execute that module might fail. In this case, you need to resubmit the request.

Composite Modules

A composite module is a module that contain zero or more submodules. Only a top-level module can contain public methods that are available for execution through the REST interface.

Submodules are similar to private modules in the following ways:

- they are not available for execution through the REST interface
- they are available for use only by other submodules and by the top-level module

Unlike private modules, which must have unique names across the system, a submodule name must be unique only for that composite module.

The top-level module of a composite module must contain DS2 source code. Submodules can contain DS2 source code or refer to a file that contains an analytic store model. Note that this is the only way to execute analytic store models.

Referencing Modules and Composite Submodules

A module can reference other DS2 packages or analytic store models.

When a module is compiled, the system is searched to satisfy these dependencies in the following order:

1. the submodules in any of the supplied module definition
2. the modules in the repository

The included submodules or modules can contain further dependencies. These are resolved in the same order as specified above.

Using Analytic Store Models

About Analytic Store Models

An analytic store file, called an ASTORE file, is a system that allows the state of a trained analytical model to be saved in a transportable form. This enables it to subsequently be used to score new data in a variety of environments. Many SAS analytical procedures save the results from the training phase of model development as analytic store models. A key feature of an ASTORE file is that it can be easily transported from one platform to another. When an analytic store is published to SAS

Micro Analytic Service, the state of the predictive model is restored and is available for scoring new data.

Publishing an Analytic Store Model

Unlike DS2 and Python modules, analytic store models are not published to SAS Micro Analytic Service as source code. Instead, analytic store models consist of binary code and metadata. Client applications deliver analytic store models to SAS Micro Analytic Service as ASTORE disk files.

Note: For information about calling an analytic store model by a DS2 module, see the next section.

Calling Analytic Store Models Using DS2

If an analytic store model has been registered with SAS Micro Analytic Service, it can be called by a DS2 module.

A DS2 module that calls an analytic store model must include an init() method that invokes the score package's setvars() and setkey() methods.

Note: Failure to set this option can cause the system to stop responding on module deletion or on shutdown.

The setvars() method is used by the DS2 score package to map variables to the analytic store model's input and output parameters. The setkey() method takes a SHA-1 hexadecimal key as input and uses it to look up the analytic store model.

SAS Micro Analytic Service automatically calls the init() method, if present, when a DS2 module is published.

Example

```
ds2_options sas;
package astoretest/overwrite=yes;
dcl package score sc();
dcl double CLAGE;
dcl double CLNO;
dcl double DEBTINC;
dcl double DELINQ;
dcl double NINQ;
dcl double VALUE;
dcl double _P_;
dcl double P__EVENT_0;
dcl double P__EVENT_1;
dcl nchar(32) I__EVENT_;
dcl nchar(4) _WARN_;
varlist allvars [_all_];

method init();
   sc.setvars(allvars);
   sc.setkey(n'EB3D1CA20AA0CB74465D25EEE2290E13692AF750');
end;

method preCode();
   _P_ = 0.999;
```

```
end;

method postCode();
end;

method term();
end;

method astoreScore(double inCLAGE, double inCLNO, double inDEBTINC,
                   double inDELINQ, double inNINQ, double inVALUE,
                   in_out double out_P_, in_out double outP__EVENT_0,
                   in_out double outP__EVENT_1, in_out nchar outI__EVENT_,
                   in_out nchar out_WARN_);
    CLAGE = inCLAGE;
    CLNO = inCLNO;
    DEBTINC = inDEBTINC;
    DELINQ = inDELINQ;
    NINQ = inNINQ;
    VALUE = inVALUE;

    preCode();
    sc.scoreRecord();
    postCode();

    out_P_ = _P_;
    outP__EVENT_0 = P__EVENT_0;
    outP__EVENT_1 = P__EVENT_1;
    outI__EVENT_ = I__EVENT_;
    out_WARN_ = _WARN_;
end;

endpackage;
```

Configuring ASTORE File System Paths

In order to publish decisions that use analytic store models to a SAS Micro Analytic Service destination, you must configure access to the location where the ASTORE files are located. Also, users who need to work with analytic store models must have Read and Write access to ASTORE directories. For more information, see "Configuring Access to Analytic Store Model Files" in *SAS Viya Administration: Models*.

Chapter 11

State Sharing between Modules

Overview **93**

Shared Vectors **94**
Overview 94
State Vector Types 94
Local State Vector Methods 96
Shared State Vector Methods 96
Setter and Getter Examples 98

Shared Hash Tables **100**
Overview 100
About Using Shared Hash Tables in DS2 100
Methods That Operate on the Default Shared Hash Table 102
Default Shared Hash Table Example 103
Methods That Operate on Non-default Shared Hash Tables 103

Overview

SAS Micro Analytic Service provides two ways to share data between modules that are executing within a user context: shared vectors and shared hash tables. *Shared vectors* are collections of data values. *Shared hash tables* are containers of stored vectors; the vectors accessed by using keys.

When it is possible to represent the data, or *state*, that you want to share across modules by a small number of vectors, the vectors can be shared with other modules by name. However, vector lookup by name is a linear search and is therefore inefficient when larger numbers of vectors are present. In such cases, shared hash tables are highly recommended because of their efficiency.

When using shared hash tables, an efficient non-cryptographic hashing function is applied to a key to quickly compute the desired vector's location within the hash table. Shared hash tables also use non-locking synchronization mechanisms to further increase efficiency.

Whether using shared vectors or shared hash tables, DS2 authors can use the MASState package to create, share, retrieve, and delete data.

Important: SAS Micro Analytic Service shared state vectors and shared hash tables are available only for DS2 modules. They are not supported for Python modules.

Important: These features support in-memory state sharing. They are not intended for state-sharing across cluster nodes.

Shared Vectors

Overview

Collections of state data fields that are managed as a unit are referred to as *state vectors*. Here are some key points about state vectors:

- A state vector contains one or more values, which are referred to by vector name and a zero-based index.
- The data values in a state vector can contain the same data types or a mix of data types.
- The number of data elements that is contained in a state vector is limited only by the available memory.
- A state vector is similar to a database record in that it can contain multiple data values of various types. However, it differs from a database record in that data values are positional, rather than organized in named columns.
- You can initialize a shared state vector from a DS2 package method, including the init() and constructor methods.
- A shared state vector name must be unique within the current user context. State vector values can have any of the following DS2 data types:
  - BIGINT
  - BINARY
  - DOUBLE
  - DOUBLE ARRAY
  - INTEGER
  - INTEGER ARRAY
  - VARCHAR
  - VARCHAR ARRAY

  Note: Binary data handling requires that you work within the limitations that are briefly discussed in a note in "Scalar Setters Example" on page 98. In SAS Micro Analytic Service, *binary data* typically refers to binary or character long objects. These can be expressed as pointer and length pairs or as character strings. Because DS2 does not support pointers directly, operations on binary data are typically performed with string manipulation functions.

State Vector Types

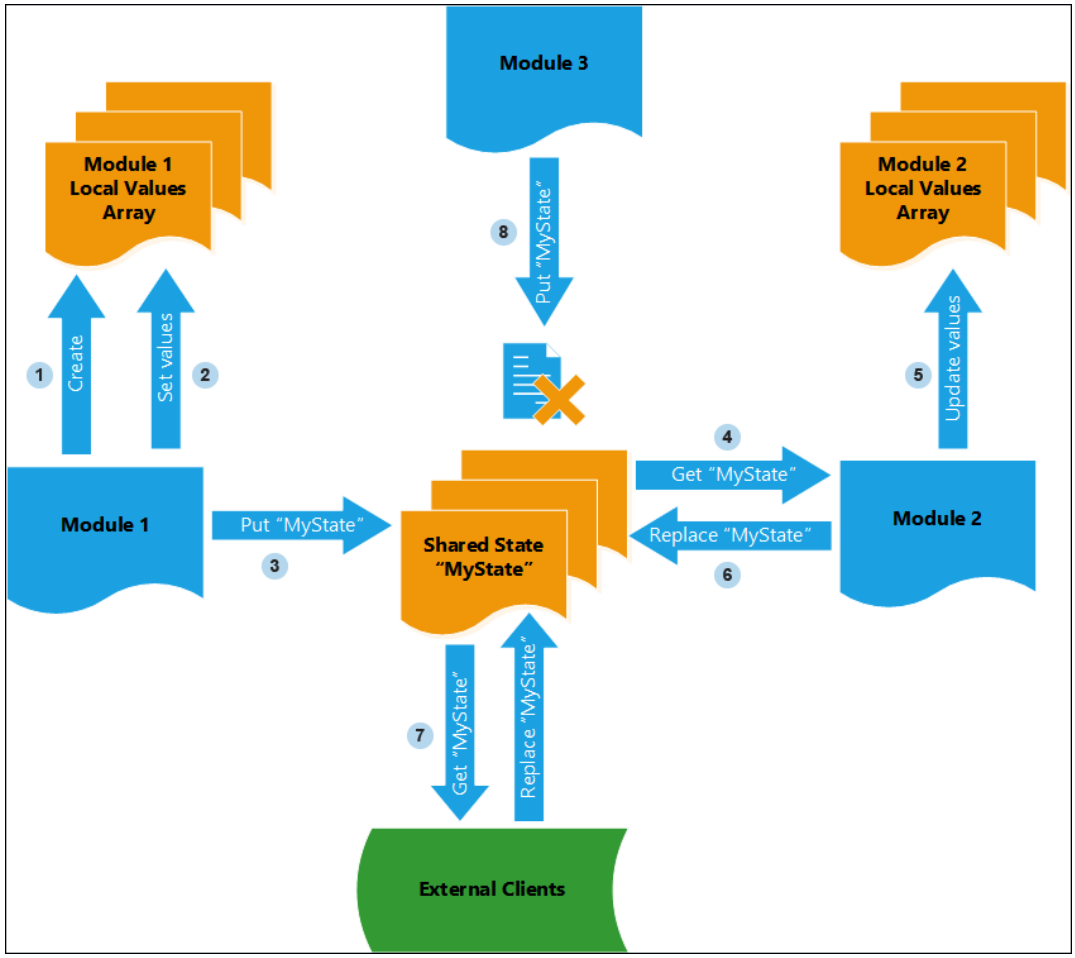
There are two categories of MASState package methods— those that operate on local state vectors and those that control state vector sharing.

Setting and retrieving individual values is always performed using local state vectors. When a shared state vector is fetched, a local copy of that vector is created and returned to the caller.

Similarly, when a state vector is shared, a copy of the local vector is created and made centrally available for fetching by other modules and transactions.

Working with local state vectors has the advantage of allowing a set of values to be updated and shared as a unit. This eliminates race conditions that could otherwise occur, and enables consistent and complete state representations.

Figure 11.1 *The State Vector Sharing Process*



1 Module 1 creates a local values array.

2 Module 1 sets the values for the array.

3 These values are published as a shared state vector and assigned the name MyState. This makes a deep copy of the vector.

4 Module 2 retrieves the MyState local vector. This makes a deep copy of the vector.

5 Module 2 updates the values and replaces the values in the local values array.

6 Module 2 replaces the values in the MyState shared state vector.

7 External clients retrieve and replace values for the MyState shared state vector.

8 Module 3 attempts to create a shared state vector called MyState. This is rejected because a shared state vector with that name already exists.

The MASState package includes 28 methods. The following sections contain usage examples for each of these methods.

Note that each example assumes that an instance of the MASState package, called st, has been created:

```
dcl package masstate st()
```

Local State Vector Methods

The following methods control the creation and deletion of local vectors.

createVector(name, size)

This method creates a local state vector with the specified name, and space for the number of values that is indicted by the specified size. The following example creates a local state vector named MyVector with a size of 4:

```
rc = st.createVector('MyVector', 4);
```

deleteVector(name)

This method deletes the local state vector referenced by name. The following example deletes the local vector created above:

```
rc = st.deleteVector('MyVector');
```

deleteAllVectors()

This method deletes all local vectors. The following example deletes all local vectors managed by the current MASState package instance:

```
rc = st.deleteAllVectors();
```

Shared State Vector Methods

The following methods control the sharing and unsharing of state vectors with other modules, and across transaction boundaries.

shareVector(name)

This method creates a copy of the named local state vector and makes it accessible to other modules within the current user context. The name passed to shareVector() must be unique within the user context. Otherwise, a duplicate name error is returned and the vector is not shared. To update an existing shared state vector, call replaceSharedVector().

```
method setValuesAndShareVector(in_out int rc);

    /* Create local vector */
    rc = st.createVector('MyVector', 4);

    /* Populate it with values*/
    rc = st.setInt('MyVector', 0, 100);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 1, 200);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 2, 300);
    if (rc ne 0) then return;
```

```
    rc = st.setInt('MyVector', 3, 400);
    if (rc ne 0) then return;

    /* Share vector with other modules */
    rc = st.shareVector('MyVector');
end;
```

fetchSharedVector(name)

This method fetches the shared state vector referenced by name and returns a local copy of it. It is used to retrieve stateful data that has been published or updated by other modules. After calling this method, the MASState package instance holds a local copy of the shared state vector, which can be referenced by name.

```
method fetchSharedVector(in_out int rc);
    rc = st.fetchSharedVector('MyVector');
end;
```

unshareVector(name)

This method removes sharing for the vector referenced by name. The shared copy of the vector is deleted from the current user context, and modules are no longer able to access it. If no shared vector with the given name exists, this is considered a valid condition and unshareVector() does not return an error. The unshareVector() method does not affect a local state vector.

```
method unshareVector(in_out int rc);
    rc = st.unshareVector('MyVector');
end;
```

replaceSharedVector(name)

This method creates a copy of the named local state vector and replaces the existing shared state vector of the same name, making the updated data accessible to other modules within the user context. The name that is passed to replaceSharedVector() must refer to an existing shared state vector. Otherwise, a **`not found`** error is returned and the data is not shared.

```
method setNewValuesAndReplaceSharedVector(in_out int rc);

    /* Populate vector */
    rc = st.setInt('MyVector', 0, 111);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 1, 222);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 2, 333);
    if (rc ne 0) then return;
    rc = st.setInt('MyVector', 3, 444);
    if (rc ne 0) then return;

    /* Share vector with other modules */
    rc = st.replaceSharedVector('MyVector');
end;
```

isVectorShared(name)

This method returns integer 1 (TRUE) if a shared state vector with the given name exists within the current user context. Otherwise, it returns integer 0 (FALSE).

```
method isVectorShared(in_out int result);
```

```
    result = st.isVectorShared('MyVector');
end;
```

Setter and Getter Examples

Setter and getter methods are provided for each data type. These methods operate on local vectors only. Individual data items are referenced by local vector name and by the zero-based index of the data value.

The examples in this section illustrate each type-specific setter method. The MASState package guards against errors such as index out of range and invalid data. As a best practice, you should check return codes, and if applicable, return them to the caller.

Scalar Setters Example

```
method testScalarSetters(varchar(32) strVal,
                         int intVal,
                         bigint longVal,
                         double dblVal,
                         bigint refVal,
                         bigint refSize,
                         in_out int rc);
    rc = -1;

    /* Populate the vector with scalars of each type */
    rc = st.setString('AllScalarsVector', 0, strVal);
    if (rc ne 0) then return;
    rc = st.setInt('AllScalarsVector', 1, intVal);
    if (rc ne 0) then return;
    rc = st.setLong('AllScalarsVector', 2, longVal);
    if (rc ne 0) then return;
    rc = st.setDouble('AllScalarsVector', 3, dblVal);
    if (rc ne 0) then return;
    rc = st.setReference('AllScalarsVector', 4, refVal, refSize);
    if (rc ne 0) then return;
end;
```

Note: setReference() accepts a bigint reference value (for example, a pointer to a BLOB or other binary data in memory) and a size (BLOB size in bytes or length of other binary data). This is due to current limitations of the DS2 BINARY data type. The getReference method returns a DS2 BINARY data type. (See "Scalar Getters Example" on page 99.) The asymmetrical nature of this setter/getter pair is due to limitations with BINARY processing that exist only on the setter side. With the exception of BINARY, all other data types are handled symmetrically.

Array Setters Example

```
method testArraySetters(varchar(32) strVal[3],
                        int intVal[3],
                        bigint longVal[3],
                        double dblVal[3],
                        in_out int rc);
    rc = -1;

    /* Populate the vector with arrays of each type */
    rc = st.setStringArray('AllArraysVector', 0, strVal);
    if (rc ne 0) then return;
```

```
    rc = st.setIntArray('AllArraysVector', 1, intVal);
    if (rc ne 0) then return;
    rc = st.setLongArray('AllArraysVector', 2, longVal);
    if (rc ne 0) then return;
    rc = st.setDoubleArray('AllArraysVector', 3, dblVal);
    if (rc ne 0) then return;
end;
```

Scalar Getters Example

```
method testScalarGetters(in_out varchar strVal,
                         in_out int intVal,
                         in_out bigint longVal,
                         in_out double dblVal,
                         in_out binary refVal,
                         in_out int rc);

    /* Retrieve scalars of each type from the vector */
    strVal = st.getString('AllScalarsVector', 0);
    if (missing(strVal)) then do;
        rc = -1;
        return;
    end;
    intVal = st.getInt('AllScalarsVector', 1);
    if (missing(intVal)) then do;
        rc = -1;
        return;
    end;
    longVal = st.getLong('AllScalarsVector', 2);
    if (missing(longVal)) then do;
        rc = -1;
        return;
    end;
    dblVal = st.getDouble('AllScalarsVector', 3);
    if (missing(dblVal)) then do;
        rc = -1;
        return;
    end;
    refVal = st.getReference('AllScalarsVector', 4);
end;
```

Note that the reference value is returned as a DS2 BINARY type, as indicated in “Scalar Setters Example” on page 98.

Array Getters Example

```
method testArrayGetters(in_out varchar strVal[3],
                        in_out int intVal[3],
                        in_out bigint longVal[3],
                        in_out double dblVal[3],
                        in_out int rc);

    /* Retrieve arrays of each type from the vector */
    st.getStringArray('AllArraysVector', 0, strVal, rc);
    if (rc ne 0) then return;
    st.getIntArray('AllArraysVector', 1, intVal, rc);
    if (rc ne 0) then return;
```

```
    st.getLongArray('AllArraysVector', 2, longVal, rc);
    if (rc ne 0) then return;
    st.getDoubleArray('AllArraysVector', 3, dblVal, rc);
end;
```

Shared Hash Tables

Overview

SAS Micro Analytic Service shared hash tables enable high-performance sharing of in-memory stateful data between modules and across transactions. Shared hash tables consist of key/value pairs, where the keys are strings and the values are state vectors. For more information about state vectors, see the previous section "Shared Vectors".

Here are some key points about shared hash tables:

- State vectors with different sizes can reside within the same shared hash table.
- Shared hash tables are visible to all modules within the same user context.
- Up to eight hash tables can exist per user context, and each hash table can contain up to 2,147,483,659 state vectors. Each state vector can contain any number of data elements.
- You can initialize a shared state vector from a DS2 package method, including the init() and constructor methods.

About Using Shared Hash Tables in DS2

The MASState package contains all the methods that are required for DS2 modules to share data across SAS Micro Analytic Service modules and transaction boundaries. These methods include operations on local state vectors and on shared hash tables.

Data can be shared among modules when you do either of the following:

- call methods that create a local state vector, populating it with values, and then putting it in a shared hash table.
- call methods that get an existing vector from a shared hash table (which makes a local copy), modifying its contents, and then replacing the vector in the hash table.

Shared hash tables are accessible by all DS2 modules within a user context.

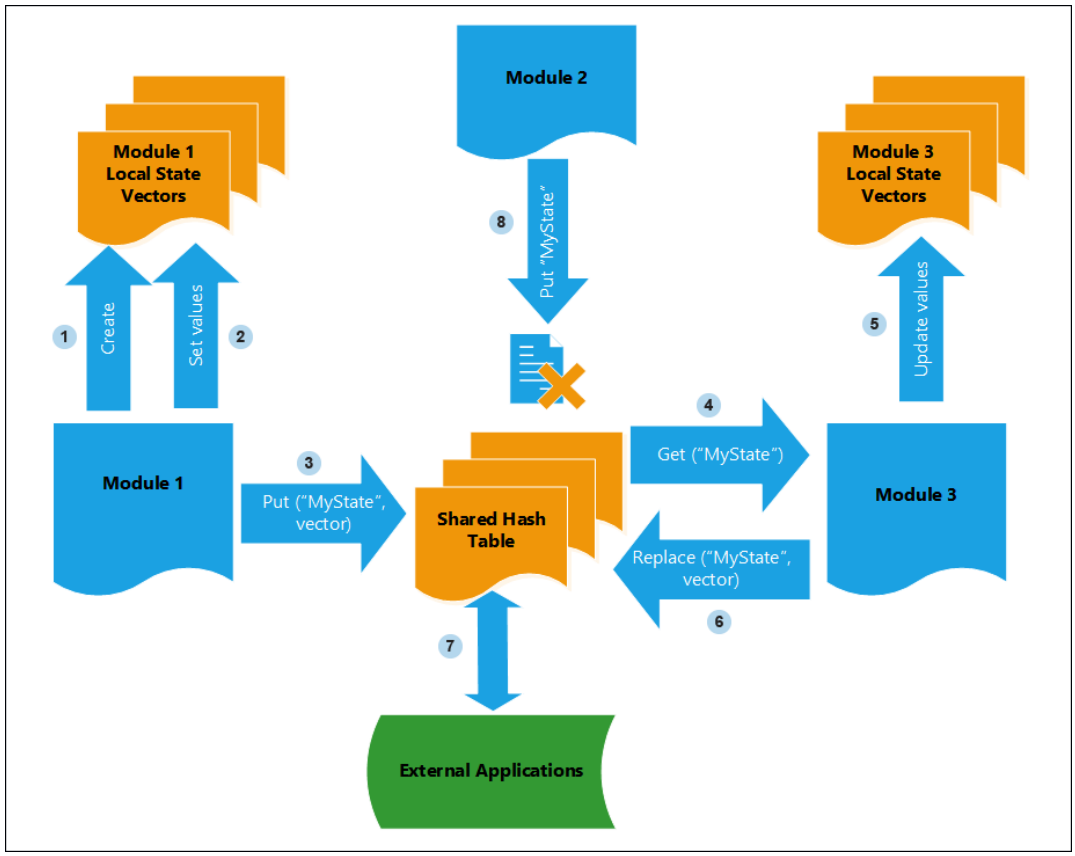
When you create a new local state vector, you assign it a name. The name must be unique within the hash table in which the vector will be stored. This name is used as follows:

- as a key when subsequently storing the vector in a shared hash table.

  That is, the name is used internally as input to a hashing algorithm that quickly computes the hash table location where the vector will be stored.
- when deleting the state vector.
- when storing or retrieving state vector data values.
- when retrieving the vector from a shared hash table.
- when replacing the vector within a shared hash table.

Up to eight shared hash tables can be defined per user context. Hash tables are referenced by index numbers zero through seven, where index zero refers to the default hash table. The default hash table is created automatically when a new user context is created. It is operated on by convenience methods that omit the table index argument. The convenience methods are clear(), isEmpty(), size(), containsKey(), put(), get(), replace(), and remove(). They are described in "Methods That Operate on the Default Shared Hash Table" on page 102.

Figure 11.2 *The Shared Hash Table Process*



1 Module 1 creates a local state vector.

2 Module 1 sets the values for the local state vector.

3 Module 1 puts these values, contained in the MyState vector, into a shared hash table.

4 Module 3 gets the MyState vector.

5 Module 3 updates the values in its local state vector.

6 Module 3 replaces the MyState state vector in the shared hash table.

7 External applications access the shared hash table to retrieve and replace the MyState state vector.

8 Module 2 attempts to store a state vector called MyState in the shared hash table. This is rejected because a state vector with that name already exists in the table.

Methods That Operate on the Default Shared Hash Table

Method Signature	Description
int clear()	Removes all state vectors from the default hash table. Returns zero if successful, and nonzero otherwise.
int isEmpty()	Returns 1 if the default hash table contains no state vectors, and zero otherwise.
bigint size()	Returns the number of state vectors currently in the default hash table.
int containsKey(key)	Returns 1 if the default hash table contains a state vector with a name matching key, and zero otherwise.
int put(key)	Inserts the state vector with the name indicated by the key into the default shared hash table. Returns zero if successful. Nonzero result codes are returned if a duplicate key exists in the default hash table or if a local state vector with a name matching key does not exist.
int get(key)	Finds a state vector in the default shared hash table with a name matching key. If found, a local copy of the state vector is made, and a zero result code is returned. If not found, a nonzero result code is returned. *Note:* If a local state vector with a name matching key already exists, and a state vector matching the key is found in the default hash table, then the existing local state vector is overwritten with the data values that are retrieved from the default shared hash table.
int replace(key)	Finds a state vector in the default shared hash table with a name matching key. If found, the state vector in the default hash table is replaced with a copy of the corresponding local state vector and a zero result code is returned. Nonzero result codes are returned if the key is not found in the default hash table, or if a local state vector with a name matching key does not exist.
int Remove(key)	Finds a state vector in the default shared hash table with a name matching key. If found, removes it and returns a zero result code. A nonzero result code is returned if the key does not exist in the default hash table.

Default Shared Hash Table Example

In the following example, method createAndPutVector() inserts a new state vector containing two integer values into the default shared hash table. Method incrementSharedValue() retrieves a state vector, named MyVector, from the default shared hash table, making a local copy. It increments the integer data value within the vector and then replaces the MyVector state vector in the default shared hash table.

```
ds2_options sas;
package statepkgtest/overwrite=yes;
dcl package masstate st();

 method createAndPutVector(varchar(32) key, in_out int rc);
   rc = st.createVector(key, 2);
   rc = st.setInt(key, 0, 100);
   if (rc ne 0) then return;
   rc = st.setInt(key, 1, 200);
   if (rc ne 0) then return;
   rc = st.put(key);
   rc = st.deleteVector(key);
end;
method incrementSharedValue(in_out int rc, in_out int int0Val);
   rc = st.get('MyVector');
   if (rc eq 0) then do;
       int0Val = st.getInt('MyVector', 0);
       int0Val = int0Val + 1;
       rc = st.setInt('MyVector', 0, int0Val);
       rc = st.replace('MyVector');
   end;
end;
endpackage;
```

Methods That Operate on Non-default Shared Hash Tables

Note: For the methods in the table, the following arguments apply:

- **`tableIndex`** indicates the hash table (0-7) on which to operate.
- **`key`** is a string value that uniquely identifies a vector within the hash table.

Method Signature	Description
int hashTblCreate(**`tableIndex`**)	Creates a new empty hash table, which can be referenced by the given table index. Returns zero if successful, and nonzero otherwise.
int hashTblDestroy(**`tableIndex`**)	Removes all state vectors from the indicated hash table, and then deletes the table. Returns zero if successful, and nonzero otherwise.
int hashTblClear(**`tableIndex`**)	Removes all state vectors from the indicated hash table. Returns zero if successful, and nonzero otherwise.

Method Signature	Description
int hashTblIsEmpty(**tableIndex**)	Returns 1 if the indicated hash table contains no state vectors, and zero otherwise.
bigint hashTblSize(**tableIndex**)	Returns the number of state vectors currently in the indicated hash table.
int hashTblContainsKey(**tableIndex**, **key**)	Returns 1 if the indicated hash table contains a state vector with a name matching key, and zero otherwise.
int hashTblPut(**tableIndex**, **key**)	Inserts the state vector into the indicated hash table at the position indicated by key. Returns zero if successful. Nonzero result codes are returned if a duplicate key already exists in the indicated hash table, or if a local state vector with a name matching key does not exist.
int hashTblGet(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key. If found, a local copy of the state vector is made, and a zero result code is returned. If not found, a nonzero result code is returned. *Note:* If a local state vector with a name matching key already exists, and a state vector matching the key is found in the indicated hash table, then the existing local state vector is overwritten with the data values that are retrieved from the hash table.
int hashTblReplace(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key. If found, the state vector in the indicated hash table is replaced with a copy of the corresponding local state vector and a zero result code is returned. Nonzero result codes are returned if the key is not found in the hash table, or if a local state vector with a name matching key does not exist.
int hashTblRemove(**tableIndex**, **key**)	Finds a state vector in the indicated hash table with a name matching key and, if found, removes it and returns a zero result code. A nonzero result code is returned if the key does not exist in the hash table.

Chapter 12

Best Practices for DS2 Programming in SAS Intelligent Decisioning

Overview **105**
Return Results **105**
Global Packages versus Local Packages **106**
Overview 106
Example of Optimized Code 106
Example of Poorly Optimized Code 106
Replacing SCAN (and TRANWRD) with DS2 Code **107**
Hash Package **109**
Character-to-Numeric Conversions **109**
Passing Character Values to Methods **109**
Performing the Computation Once **110**
Moving Invariant Computations Out of Loops **110**

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.

Return Results

If a DS2 method, or any method it calls, can result in a status code or failure, always include a method output argument for returning the result to the caller.

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 the CPU load due to 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 13

Python Support in SAS Micro Analytic Service

Introduction **111**

About Creating Python Modules **112**

Public and Private Methods **113**
Overview 113
About Private Methods 113
About Public Methods 114
Return Values 114
Examples: Public and Private Methods 115

Working with Python and SAS Micro Analytic Service **116**

Compiling Python Modules **117**

Configuring Python for SAS Intelligent Decisioning **118**
Environment Configuration 118

Introduction

SAS Micro Analytic Service supports modules that are written in the Python programming language. A Python module represents a group of related Python functions.

Input arguments are given in the function's argument list. The objects, variables, and expressions listed in a Python function's return statement are positional with respect to the output variables.

The output variables are listed in the function’s "Output:" docstring that is specified in the first statement of the function. Any method that includes the "Output:" docstring is considered a public method. Otherwise, it is considered a private method. For information, see the sections later in this chapter.

Input and output argument names live in a single namespace and therefore cannot be the same. This means that update arguments are not supported. This is true for all module types in SAS Micro Analytic Service, even though the Python language does not enforce such a restriction.

Here is an example of a Python public function that can be hosted by SAS Micro Analytic Service.

```
import sys
import math
```

```
import pandas as pd
import numpy as np
def nppd(a):
    "Output: ser1"
    npa = np.array([[1,2,3],[4,5,6]])
    ser1 = pd.Series([212, a, -273])
    return ser1.tolist()

def trucks(Eng_Load, Oil_Temp, Eng_RPM):
    "Output: ser1, x, syspath"
    inputs = pd.Series([Eng_Load, Oil_Temp, Eng_RPM])
    b = np.arange(100)
    number = 0
    for index, item in enumerate(inputs):
        number += item + b[index + 7]
    # is it even or odd?
    x = math.fmod(number, 2)
    return nppd(Oil_Temp), x, getsyspath()

def getsyspath():
    "Output: p"
    p = [None] * 50
    # print(sys.path)
    syspaths = sys.path
    i = 0
    for path in syspaths:
       p[i] = path
       i = i + 1
    return p
```

Here is an example of a Python public function that has input arguments a and b, and no output.

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

After Python is configured, see Appendix 1, “Executing Python Modules in DS2 Modules,” on page 139 for additional information.

About Creating Python Modules

There are two ways that you can create Python modules in SAS Micro Analytic Service:

- Create a DS2 module that uses the DS2 PyMAS package to publish the module.

  This is useful when you intend to use a Python module from other DS2 modules. For information, see Appendix 1, “Executing Python Modules in DS2 Modules,” on page 139.

- Directly create a Python module using the REST interface without using any DS2 code.

In this case, the **`type`** field of the module definition JSON is set to *text/x-python*, and the source contains the Python code. This is useful when you need SAS Micro Analytic Service to execute only Python code.

See the REST API documentation for details about how to create the module definition JSON representation.

Both approaches support the same data types and require that the Python code comply with the same rules and restrictions.

Note the following information about the two Python approaches:

- Here are key differences that you should be aware of:
  - When you use the PyMAS package, the Python code is compiled during the execution of the module. It is important to check any error codes that are generated from the publish call before you execute the code.
  - When you use Python modules directly, the Python code is compiled (using the SAS Micro Analytic Service REST service) when the module is created or updated.
- Python modules are stand-alone—submodules are not supported and inter-module dependencies are not handled.
- Python modules can use other Python modules that have been published to SAS Micro Analytic Service. A DS2 PyMAS package instance can use the useModule method to specify a previously published module instead of publishing a new revision of the Python module.

Public and Private Methods

Overview

SAS Micro Analytic Service enables the use of hosting public and private methods, where a method is a Python function. Note that public and private methods are SAS Micro Analytic Service concepts, and are not Python features.

In general, any method that includes the "Output:" docstring is considered a public method. If a method does not have the "Output:" docstring, then it is considered a private method. However, there are syntax requirements that must be followed for the docstring and the output arguments. For details, see “About Public Methods” on page 114.

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.

About Private Methods

Here are details about using a private method:

- A private method can be called internally by other methods (either public or private).
- A private method cannot be called directly (externally).
- Private methods are useful when used as utility functions within a package.

About Public Methods

Here are details about using a public method:

- For a function that has at least one output argument, there must be a space between "Output:" and the first output argument name. For examples, see the next section "Examples: Public and Private Methods" on page 115.
- When there is more than one output argument, the output argument names must be comma separated.
- Line two of the function must begin with a docstring, and the first non-whitespace token must be "Output:".
- All public functions that return 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:" docstring to indicate a public function. It should simply be "Output:", with no output arguments listed.
- Order does matter. Therefore, the order in the return statement must match the order in the "Output:" line. A best practice is to copy and paste from one to the other.

Return Values

When Python integers are returned, they are converted to int64_t. If the Python integer is too large for the int64_t type, a run-time error is reported.

Although Python packages might support abstract types that provide an additional layer on top of the built-in scalar types, such as integer and float, they cannot be returned from SAS Micro Analytic Service public functions. Packages that have these abstract types provide functions that can be used to extract the built-in type. Here is an example:

```
return numpy.float(x), numpy.int(y)
```

Note the following information about the return statement for Python functions:

- Functions that return nothing do not require a return statement. Python returns the NoneType object. Therefore, an empty return statement (**`return`**) and returning None (**`return None`**) are equivalent.
- Functions that return a single argument can return None. SAS Micro Analytic Service maps the None value to the specific Missing value for the expected static type.
- Functions that return multiple values can be formatted with or without parentheses (for example, **`return a,b,c`** or **`return (a,b,c)`**).

  Note: An attempt to return None from a function that expects multiple values is invalid.

When return values are transferred between Python and SAS Micro Analytic Service, the following rules are enforced by SAS Micro Analytic Service:

- A singleton value (non-tuple) is mapped to a tuple with a single value.
- Tuples map to multiple return arguments.

- Arrays and tuples cannot be used interchangeably. The Python tuple type is used to return multiple values. The Python list type is used for SAS Micro Analytic Service arrays.
- The argument count must match.

The following code sample and table show some examples that illustrate the application of these rules.

```
def one(a):
"output: b"
return oneVal

def two(a,b)
"output: a,b"
return twoVal
```

Python Value Assignment	SAS Micro Analytic Service Value Assignment
`oneVal = 1`	`oneVal = (1)` (one integer value)
`oneVal = 1,`	`oneVal = (1)` (one integer value)
`oneVal = (1,)`	`oneVal = (1)` (one integer value)
`oneVal = (1, 2)`	An error occurs. It indicates that the expected output variables do not match the actual output variables.
`twoVal = 1,2`	`twoVal = (1,2)` (two integer values)
`twoVal = (1,2),`	An error occurs. It indicates that an unknown tuple type is present.
`twoVal = ([1],[2])`	`twoVal = ([1],[2])` (two integer arrays)

Examples: Public and Private Methods

As mentioned previously in this chapter, for a method to be public, the output variables must be listed in the function's "Output:" docstring that is specified on the first statement of the function. This is the second line of the method, immediately following the "def" line.

Here are some examples. Note that the fun2 function would be considered private because the docstring does not begin with "Output:".

```
def fun1( a, b ):
    "Output:"
    """ This is a public function,
```

```
        but has no output args."""
    return

def fun2( a, b ):
    """This will be private since the docstring doesn't begin with Output:
      Output: x, y, z"""
    return a+2, b*4, a/b

def fun3( a, b ):
    '     Output: x, y, z'
    ''' multi
        Line
        doc string'''
    return a+2, b*4, a/b

def fun4( a, b ):
    '''  Output: x, y, z'''
    """ multi
        Line
        doc string"""
    return a+2, b*4, a/b

def fun5( a, b ):
    '''Output: q, r, s,
               t, u,
               v,
               w'''
    return a+2, b*4, a/b
```

Working with Python and SAS Micro Analytic Service

When you work with Python and SAS Micro Analytic Service, note the following:

- Multi-tenant deployments are not supported.
- When sharing modules across other modules, because all modules belong to the same tenant, you should understand the following logic:

  If module A and module B each import module C, updates to module C affect both module A and module B.
- All client requests are passed to SAS Micro Analytic Service with an OAuth access token. This token is not passed to the Python subprocess. Therefore, Python code cannot connect to a SAS service that requires an access token.

  Note: This is a SAS Micro Analytic Service limitation on Python modules.
- SAS Micro Analytic Service Python interface usage is prohibited when any of the following situations exist:
  - The application enabled MASHostAccessDeny when initializing SAS Micro Analytic Service.
  - The MASHostAccess=DENY environment variable is set.
  - The application provides MASHostOSID credentials that fail to authenticate.

  - The CAS session is not running under a host operating system identity and it attempts to run an action that uses SAS Micro Analytic Service.

- In this release of SAS Micro Analytic Service, maintaining multiple revisions of a Python module is not supported. A subsequent publish request on the same Python module context replaces the existing revision. This means that the revision number always remains at 1.

  For example, consider the following two Python modules:

  test.py:

  ```
  def execute(a,b):
     "Output: c"
     c = (a+b)
     return c

  def score(x):
     "Output: y"
     return x * 0.5;
  ```

  test2.py:

  ```
  def execute(a,b,c):
     "Output: d"
     d = (a+b+c)
     return d;
  ```

  If you publish test.py and then publish test2.py to the same module context, the first execute function is replaced by the second one. The new execute function has an additional input argument (c) and a different output argument (d). Also, the new revision no longer includes the score function.

- The SAS Micro Analytic Service Python interface usage is prohibited by default when a SAS session is in lockdown mode.

  To enable it, you can use the LOCKDOWN statement with the ENABLE_AMS argument on the server. Set the argument value to *PYTHON*. If you want to use the DS2 PyMAS package, you also need to set the *PYTHON_EMBED* argument value. For example:

  ```
  LOCKDOWN ENABLE_AMS=PYTHON ENABLE_AMS=PYTHON_EMBED
  ```

  For more information, see "Configuring SAS to Run External Languages" in *SAS Viya Administration: Programming Run-Time Servers*.

- SAS Micro Analytic Service applications can specify a host operating system account to use as the owner of Python subprocesses that are launched when using SAS Micro Analytic Service modules. For details, see the next section, "Compiling Python Modules".

Compiling Python Modules

A Python module is compiled and run in its own process. For security reasons, you might need to limit the access of Python processes to the file system or other resources of the host server. You can accomplish this by using SAS Viya external credentials functionality. The account associated with the credentials that you specify should have a scope that is valid for running Python processes.

It is important to understand that the account that you configure to run Python processes is not a SAS Viya account. Rather, it is a host server account that you specify using the external credentials functionality in SAS Environment Manager. For more information, see "Manage Credentials" in *SAS Viya Administration: External Credentials*.

For SAS Micro Analytic Service to launch Python processes using this account, you use SAS Environment Manager to configure the following properties:

- **core.mashostuser**: specifies the user ID that is associated with the account
- **core.mashostdomain**: specifies the Authentication domain that is associated with the account

You must re-start SAS Micro Analytic Service to enable the use of the specified credentials.

For more information about configuring these properties, see "supplementalProperties" on page 129.

If you do not configure these properties, Python processes are run under the account of the SAS Micro Analytic Service process.

Configuring Python for SAS Intelligent Decisioning

Environment Configuration

SAS Micro Analytic Service does not require a specific version of Python. However, it is possible that Python code that you publish to SAS Micro Analytic Service might have dependencies to a specific version of Python or Python packages. To configure Python for use within SAS Micro Analytic Service, note the following requirements:

- The MAS_PYPATH environment variable must be set. It specifies the absolute path to the Python executable file. This path must be the same on all machines where the SAS Micro Analytic Service REST API is deployed. Here are some examples:
  - UNIX platform:

    ```
    MAS_PYPATH=/usr/bin/python
    export MAS_PYPATH
    ```

  - Windows platform:

    ```
    set MAS_PYPATH=c:\python\python.exe
    ```

    Note: You are prompted to supply this value during the installation and configuration of SAS Micro Analytic Service. Therefore, it should be configured for you following that process. However, if the value was not specified at that time, you must add it to the wrapper.conf (Windows platform) or setenv.sh (UNIX platform) file. If you have other Python configuration commands that are required by your Python distribution, you should also add those.
- When SAS Micro Analytic Service launches the Python client, it waits for the environment to reconnect. By default, this wait time is 30 seconds. To change the wait time, you can set the MAS_PYWAIT environment variable and assign the applicable value (in milliseconds).
- Note the following important information about using Python:

  - You must add the environment configuration commands to the following file:

    /opt/sas/viya/config/etc/sysconfig/microanalyticservice.conf

    If this file does not exist, you must create it.

  - 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 -*-
    ```

  - 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"
    ```

  - Python 3.*x* uses UTF-8 as the default encoding. Therefore, the encoding issues noted above 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.

- For Python environments prior to version 3.7, ensure that the LANG environment variable is set as follows:

  ```
  LANG=*.UTF-8
  ```

  The asterisk (*) corresponds to the applicable locale value for your environment. Some examples are *en_US.UTF-8* (English United States), *fr-FR.UTF-8* (French France), or *de-CH.UTF-8* (German Switzerland).

  Note: If the LANG environment variable is not defined, SAS Micro Analytic Service sets its value to **en_US.UTF-8**.

- Note the following information when you set environment variables (such as PYTHONPATH) in the sub-processes that are launched to run on Python:

  - In a UNIX environment, exported shell environment variables are inherited by the sub-processes.

  - In a Windows environment, you must define system and user environment variables in **Control Panel** ⇨ **System and Security** ⇨ **System** ⇨ **Advanced system settings** ⇨ **Environment Variables**.

Chapter 14

Administration

SAS Micro Analytic Service Administration **121**

Database Access with DS2 . . . . . **122**
Architectural Considerations 122
Connection Strings and Configuration 122
Using Third-Party Database Drivers 123

Starting and Stopping SAS Micro Analytic Service **124**

Synchronous, Asynchronous, and Timed Execution **124**

SAS Micro Analytic Service Configuration **125**
Overview 125
jvm 125
logging.level 126
sas.microanalyticservice Sections 126
supplementalProperties 129

SAS Micro Analytic Service Logging **131**
Overview 131
Loggers and Logging Levels 131

SAS Micro Analytic Service Security and Authorization **133**

Secure DS2 HTTP Package Usage **134**

Moving Objects by Using the SAS Viya Transfer Service **134**

SAS Micro Analytic Service Administration

SAS Micro Analytic Service is implemented as a SAS Viya microservice. Most of the administrative tasks and capabilities are described in *SAS Viya Administration: Orientation*.

Because SAS Micro Analytic Service is implemented using both Java and SAS threaded kernel technology, there are some areas in which it differs from a standard SAS Viya microservice.

The items that are implemented using SAS threaded kernel technology are referred to as the Micro Analytic Service core component. The items that are implemented using Java are referred to as the REST service.

Database Access with DS2

SAS Micro Analytic Service supports database I/O through the DS2 SQLStmt package. Supported databases include DB2, Oracle, Postgres, SQL Server, and Teradata.

Architectural Considerations

If the SQLSTMT package is used to access third-party databases, the associated SAS/ACCESS products and third-party client libraries are required on each system that is running SAS Micro Analytic Service. For more information, see the SAS Viya deployment documentation on support.sas.com.

Vendor libraries are provided by the database vendor, such as IBM (for DB2) or Oracle Corporation (for an Oracle database). They must be installed separately according to the instructions that are provided by the vendor. In addition, SAS Micro Analytic Service must be configured to use these vendor libraries by setting appropriate environment variables. For information, see "Using Third-Party Database Drivers" on page 123.

SAS Micro Analytic Service is a multi-threaded service in which each thread requires concurrent access to the database. Access to SAS data sets is supported. However, since SAS data sets use file-level locking, they are not suitable for writing from multiple threads. It is recommended that you carefully set appropriate connection options before reading SAS data sets from multiple threads, since problems can lead to a deadlock situation. For these reasons, the use of a third-party database management system is highly recommended.

Connection Strings and Configuration

Overview

Connection strings are used to specify database connection information such as host, port, driver, database, catalog, schema, credentials, and options. The SQLSTMT package supports the FedSQL dialect. Therefore, the connection string should begin with the following information:

```
DRIVER=SQL;CONOPTS=(
```

In the above string, **SQL** specifies the FedSQL language driver as the managing driver, and one or more target driver connection strings are specified within the CONOPTS= option.

The following example illustrates a federated connection string that includes Oracle and PostgreSQL data sources:

```
driver=sql;conopts=((driver=oracle;catalog=acat;uid=scott;
pwd=tiger;path=oraclev11.abc.123.com:1521/ORA11G);
(driver=postgres;catalog=bcat;uid=myid;pwd='mypass';
server=sv.abc.123.com;port=5432;DB=mydb;schema=public))
```

Passwords can be encoded values that are produced by PROC PWENCODE. For more information about connection string structure, see the SQLSTMT package information in *SAS DS2 Language Reference* and Appendix 4, "Table Service Driver Reference," on page 167.

Defining and Configuring Connection Strings

SAS Micro Analytic Service allows the connection string to be supplied through a configuration process. Although the SQLSTMT package allows specifying a connection string in code, SAS recommends that it be supplied using the SAS Micro Analytic Service configuration. The benefits to doing this are as follows:

- When connection information is not included in the code, the module is more portable. This enables it to be moved to other SAS Micro Analytic Service deployments without requiring code modifications. This can help implement strategies for environment promotion (Development-Test-Production) as well as for hot standby failover.
- When SAS Micro Analytic Service manages the database connection, it can detect whether connectivity to the database is lost, and automatically tries to reconnect on a periodic basis.
- If severe errors occur during database access, SAS Micro Analytic Service automatically attempts to recover from the error by recompiling the DS2 packages.

The connection string for SAS Micro Analytic Service deployments is configured by setting the sas.microanalyticservice.service.connectionstring property in SAS Environment Manager. For more information, see "core" on page 126.

This property is configured on a per-tenant property basis. This means that every tenant has its own private copy of the property. It is not shared with other tenants.

Note: Although only one connection string can be specified per tenant, the connection string itself can be federated, which allows multiple databases to be accessed using catalog and schema names.

Using Third-Party Database Drivers

As mentioned in the "Overview", database vendor-specific libraries (often referred to as database driver software) must be installed according to the instructions that are provided by the database vendor. As part of the installation, these drivers provide either manual instructions or scripts that set up environment variables (for example, PATH and LD_LIBRARY_PATH). To use these libraries, client software such as SAS Micro Analytic Service must have access to these environment variables.

To enable access to these environment variables, edit the following file to include the definitions of the required environment variables:

`/opt/sas/viya/config/etc/sysconfig/microanalyticservice.conf`

In the path above, *viya* is the deployment ID.

Important: This file does not exist in a standard installation and must be created. Each node of a cluster deployment that has a deployment of SAS Micro Analytic Service must be updated with the third-party driver installation and this file.

After installing the drivers and updating the file with the appropriate values, you must restart SAS Micro Analytic Service to apply the changes.

TIP To set this configuration, it can be helpful to view all environment variables that are available to SAS Micro Analytic Service. To accomplish this, enable TRACE logging for com.sas.mas.impl.MASFactoryImpl.

SAS Micro Analytic Service enables access to HTTP and HTTPS web services through the DS2 HTTP package. This package can execute HTTP requests to, and receive responses from, HTTP and HTTPS web services. Direct file I/O is not supported. As a result, DS2 hash packages cannot be populated from the contents of a file. For more

information, see *SAS DS2 Language Reference* and *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

Starting and Stopping SAS Micro Analytic Service

To start or stop SAS Micro Analytic Service, use the service **start** or **stop** commands as described in *SAS Viya Administration: General Servers and Services*.

The following commands are examples that are specific to SAS Micro Analytic Service. In these examples, *viya* is the deployment ID and *default* is the instance ID.

These commands start and stop the service:

```
sudo service sas-viya-microanalyticservice-default start
```

```
sudo service sas-viya-microanalyticservice-default stop
```

This command verifies whether the service is running:

```
sudo service sas-viya-microanalyticservice-default status
```

Note: For a multi-machine deployment, you must start and stop the service on each deployment server.

Synchronous, Asynchronous, and Timed Execution

A SAS Micro Analytic Service module contains one or more steps. Each step is executed by passing input parameter values to the server.

By default, this execution is processed synchronously. This means that the client makes the SAS Micro Analytic Service REST call, and the call returns when the execution is complete. Upon successful execution, the **outputs** variable in the reply contains the output of the execution, and the **executionState** variable is assigned the value **completed**.

Depending on your requirements, it might be beneficial for SAS Micro Analytic Service to use asynchronous execution. When you use asynchronous execution, the input parameter values are passed to the server and the execution occurs in a separate thread. Any errors in execution are logged on the server. If logging is set at the TRACE level, the result of the execution is also logged.

To use asynchronous execution, include the **waitTime** query parameter with an assigned value (in milliseconds). Here is an example of a POST request that includes the **waitime** query parameter:

```
POST https://www.example.com/microanalyticScore/modules/{moduleId}/steps/{stepId}?waitTime=6000
```

When you include the **waitTime** query parameter, the **executionState** variable with an assigned value is included in the reply. Here are the possible values:

- **completed**: the execution completed within the specified **waitTime** value. The result is returned.
- **timedOut**: the execution did not complete within the specified **waitTime** value. The **outputs** variable in the reply is empty.

- **submitted**: the assigned **waitTime** query parameter value is 0 (zero). The **outputs** variable in the reply is empty.

  TIP Assigning **waitTime=0** is useful when a call does not generate output (for example, when executing a step that fires an event).

SAS Micro Analytic Service Configuration

Overview

You use SAS Environment Manager to configure SAS Micro Analytic Service Score service property settings. SAS Environment Manager is a web application for managing a SAS Viya environment. It includes a dashboard view, which provides a quick overall look of your environment's health and status, as well as detailed views that enable you to examine and manage your environment in detail. For more information, see *SAS Viya Administration: Using SAS Environment Manager*.

In SAS Environment Manager, the SAS Micro Analytic Service Score service configuration parameters are organized into sections. Here are the key configuration sections:

- **jvm**
- **logging.level**
- **sas.microanalyticservice.properties** and **sas.microanalyticservice.system**
- **supplementalProperties**

The following sections describe the properties that are contained in each section.

jvm

The **jvm** section contains the properties to configure the Java virtual machine to run SAS Micro Analytic Service.

Important: If you change the value for any of these properties, you must restart SAS Micro Analytic Service for the change to take effect.

Property	Type	Default Value	Description
java_option_xmx	String	-Xmx1024m *Note:* This represents 1GB.	The Java maximum heap space size. *Note:* You might need to increase this value depending on the number and size of the modules.
java_option_xss	String	-Xss512k *Note:* This represents 512K.	The Java thread stack size. *Note:* The assigned value should not be less than 512K.

Property	Type	Default Value	Description
java_option_xms	String	-Xms1024m *Note:* This represents 1GB.	The Java minimum heap space size. By default, this property does not appear SAS Environment Manager. You can add it by using the **Add property** option in the **jvm** section. *Note:* You might need to increase this value depending on the number and size of the modules. It is recommended that you do not assign a value smaller than 1GB.

logging.level

There are multiple logging.level sections, each that sets the logging level for a specific area of functionality. For information, see "Logging Levels" on page 133.

sas.microanalyticservice Sections

The **sas.microanalyticservice.system** section contains items that are specific to system-wide configuration of SAS Micro Analytic Service. The **sas.microanalyticservice.properties** section contains tenant-specific items for use in a multi-tenant deployment. Configuration items that are tenant-specific are noted as such in their description. Each sas.microanalyticservice section contains configuration items organized by the following sub-sections, as applicable.

asynchronousexecution

This section contains properties that configure asynchronous execution.

Property	Type	Default Value	Description
corepoolsize	Integer	5	The number of core worker threads for asynchronous requests.
maxpoolsize	Integer	10	The number of maximum worker threads for asynchronous requests.
queuecapacity	Integer	100	The maximum number of asynchronous requests that can be queued for execution.

core

This section contains properties that affect the Micro Analytic Service core component.

Important: Unless otherwise noted, if you change the value for any of these properties, you must restart SAS Micro Analytic Service for the change to take effect.

Property	Type	Default Value	Description
connectionstring	String		The FedSQL connection string used to connect to a database. *Note:* This item is tenant-specific. Therefore, it is configured in the **sas.microanalyticservice.properties** section. *Note:* You do not need to restart SAS Micro Analytic Service if you change the value that is assigned to this property.
dbconnretries	Integer	10	The number of times that the system attempts to connect to a database.
dbconnretryintervalseconds	Integer	5	The amount of time, in seconds, that the system waits before retrying to connect to a database.
ds2maxrecompilecount	Integer	0	If there is an error because of failing database connections, the maximum number of times that the system tries to recompile DS2 code before ejecting the module.
gcintervalseconds	Integer	60	The time, in seconds, between garbage collection runs.
graceperiodseconds	Integer	10	The amount of time to wait, in seconds, before cleaning up the assets that are associated with a deleted revision. *Note:* A request to execute a deleted revision, even during the grace period, is rejected with an error message that indicates the revision has been deleted.
nativebuffersize	Integer	512	The size of the buffer, in bytes, to exchange data between the REST service and SAS Micro Analytic Service core.
numthreads	Integer	4	Specifies the number of threads in the SAS Micro Analytic Service core. The amount of memory that is used by SAS Micro Analytic Service core is determined by the number of threads. Note the following information when setting a value: • The compilation time for modules is proportional to the number of threads. • You can increase or decrease the value, depending on the availability of system memory. • Setting the value to 0 causes the SAS Micro Analytic Service core to use all possible cores on the system. For a system in which SAS Micro Analytic Service is the primary application, this is recommended.

historyharvester

This section contains the properties that are associated with the extraction and publishing of the history record tasks map.

Property	Type	Default Value	Description
maxrecordsinmessage	Integer	100	The maximum number of history records that are included in a message. *Note:* This property is tenant-specific. Therefore, it is configured in the **sas.microanalyticservice.properties** section.
ratemilliseconds	Integer	5000	The number of milliseconds to wait between extracting history records. *Note:* This property is tenant-specific. Therefore, it is configured in the **sas.microanalyticservice.properties** section.

historyscheduler

This section contains a property that configures the number of threads available for scheduling the extraction and publishing jobs.

Property	Type	Default Value	Description
poolsize	Integer	5	The number of threads available for scheduling the extraction and publishing jobs.

service

This section contains the properties that configure the REST service component.

Property	Type	Default Value	Description
defaultmasuserctxname	String	defaultMASUserCtxName	The default value of the user context name, if the tenant name is not used.
moduleidgeneration			A sub-category containing two properties that control how the module ID is determined. The properties are factorytype and forcelowercase.
factorytype	One of the following: • GUID • ModuleName • ModuleNameOverridePackage • PackageName	ModuleNameOverridePackage	This value determines how the module ID is created for a new module. The module name is the one that is passed to the system during module creation.
forcelowercase	Boolean	True	If this property is set to True, the module ID returned is always lowercase.

Property	Type	Default Value	Description
sasmode	Boolean	True	If this is set to True, the DS2 code is compiled with ds2_options sas.

supplementalProperties

This section contains optional properties that, by default, do not appear in SAS Environment Manager. When you add them, they appear in the **supplementalProperties** section. For information about creating additional property configuration instances, see *SAS Viya Administration: Using SAS Environment Manager*.

Property	Type	Default Value	Description
core.mashostuser	String	Not applicable	The user ID for Python processes to use to access the host server. If not specified, Python processes are run under the account of the SAS Micro Analytic Service process. For information, see "Manage Credentials" in *SAS Viya Administration: External Credentials*.
core.mashostdomain	String	Not applicable	The Authentication domain to use to authenticate the user ID that is specified for core.mashostuser. If not specified, the Python processes are run under the account of the SAS Micro Analytic Service process. For information, see "Manage Credentials" in *SAS Viya Administration: External Credentials*.
core.minfreememoryfloor	Integer	500000000	The minimum amount of memory, in bytes, that is allocated to compile a module. The default value is equal to 500MB.
core.tktstacksizekbytes	Integer	8192	Sets stack size, in kilobytes, for worker threads in SAS Micro Analytic Service core. The default value is sufficient for most purposes. You might need to increase the value to compile large DS2 packages that contain numerous (for example, more than 5,000) IF-THEN/ELSE statements. *Important:* If you change the value for this property, you must restart SAS Micro Analytic Service for the change to take effect.
core.profilesamplefrequency	Integer	2000	Used in conjunction with profile logging. Specifies the sample rate to log the execution time of the module code.

Property	Type	Default Value	Description
service.cache.refresh.ratemilliseconds	Integer	10000	Specifies the rate, in milliseconds, at which the database is monitored for changes, in microseconds. The default value is equal to 10 seconds. This is relevant only for a clustered deployment that contains multiple SAS Micro Analytic Service nodes. Note the following information when setting a value: • For a system to which few changes will be made, this parameter can be set to a very large value, such as 30 minutes or more. An execution-only system is an example of a system with infrequent updates. • For a system to which many updates will be made, this parameter can be set to a lower value, such as 5 seconds.
service.alwayscheckdatabaseonexecute	Boolean	False	Specifies whether the execute call checks the database for the latest copy of the module, before every invocation. This is relevant only for a clustered deployment that contains multiple SAS Micro Analytic Service nodes. Note the following information when setting a value: • The default value (False) results in efficient execution. • Set the value to True if modules are updated frequently. It is important to execute the latest copy in a clustered deployment.
service.removetrailingunderscoresfrominputs	Boolean	False	SAS Micro Analytic Service requires separate parameters for input and output. However, DS2 does not allow the same parameter to be listed twice for a method. To ensure that this does not happen, SAS Intelligent Decisioning generates code in which input variables are named with a trailing underscore (_). You can use this property to manage this behavior. This property specifies whether the trailing underscore is retained (False) or discarded (True) from the input variable names in the following situations: • when returning the list of parameters for a method through the REST interface • when accepting input for the execute call You might need to set this to True if your site uses a repository of data items with which all applications must comply.
service.timeouts.maxloadwaitexecmillis	Integer	10000	Specifies the time-out value, in milliseconds, for execution operations. The default value is equal to 10 seconds.
service.timeouts.maxloadwaitnonexecmillis	Integer	120000	Specifies the time-out value, in milliseconds, for query operations. The default value is equal to 2 minutes.

Property	Type	Default Value	Description
service.timeouts.maxmodulecompiletimemillis	Integer	600000	Specifies the time-out value, in milliseconds, when compiling DS2 modules. The default value is equal to 10 minutes. This time-out value applies to both asynchronous and synchronous compilation requests.

SAS Micro Analytic Service Logging

Overview

The SAS Micro Analytic Service core and the REST service each create log files. A new log file is created according to the resource logging configuration. This might include the occurrence of any of the following:

- a new day begins (rollover at midnight)
- the log file size exceeds the maximum value
- the service starts

The SAS Micro Analytic Service log files are typically created in the following folder:

`/opt/sas/viya/config/var/log/microanalyticservice/default`

In the path above, *viya* is the deployment ID, and *default* is the instance ID.

The following table provides default logging information for each service:

Service	Default naming convention	Default file rollover configuration
SAS Micro Analytic Service core	sas-microanalyticservice-core_yyyy-MM-dd_HH-mm-ss.log	When either of the following occur: • the log file size exceeds 100K • the service starts
REST	sas-microanalyticservice_yyyy-MM-dd_HH-mm-ss.log	each day at midnight

For complete information about logging, see *SAS Viya Administration: Logging*.

Loggers and Logging Levels

Loggers are split into two groups: those that apply to SAS Micro Analytic Service core and those that apply to the REST service.

SAS Micro Analytic Service Core

Logger names that start with the following prefixes apply to the SAS Micro Analytic Service core: Admin, App, Audit, Perf.

Here are some important SAS Micro Analytic Service core loggers:

Name	Events Logged	Default Value
App.tk.MAS	Method execution events	INFO
App.tk.MAS.Service	Start-up and shutdown events and the SAS Micro Analytic Service version number	INFO
App.tk.MAS.Python	Python events	INFO
App.tk.MAS.CodeGen	Compilation messages produced during an attempt to publish. When a publish request fails, error information is logged regardless of the App.tk.MAS.CodeGen logger level.	FATAL
App.SQLServices.license App.license DataFlux.Licensing	Licensing events	ERROR
Audit.Table.Connection	Database connection events	INFO

Loggers that start with App.TableServices.DS2.Runtime. or App.TableServices.DS2.Config. can be used to diagnose DS2 problems. Module code might also use other loggers.

When diagnosing DS2 problems, it is important to note that the App.TableServices.DS2.Runtime.* and App.TableServices.DS2.Config.* loggers do not inherit configuration from their ancestors. They must be configured explicitly, if you want to capture logging events that are directed to those loggers. It is recommended that you configure them only when diagnosing a DS2 problem since the additional logging traffic affects performance. For more information about those DS2 loggers, see the "DS2 Loggers" section of *SAS DS2 Programmer's Guide*.

Detailed information about operations such as compilation start and finish is logged at the DEBUG level. Warning and error conditions are logged at the WARN or ERROR levels, as appropriate.

Note: When App.tk.MAS.Service logger's level is set to DEBUG or TRACE, you see a message logging event that provides the SAS Micro Analytic Service version number in the log. Here is an example:

```
May14 15:12:49 [00000007] DEBUG App.tk.MAS.Service -
Micro Analytic Service version information: 5.3, V.03.04M0P05052019,
Mon May  6 03:00:21 EDT 2019
```

Important: If you change the value for any SAS Micro Analytic Service core loggers, you must restart SAS Micro Analytic Service for the change to take effect.

REST Service

Logger names that start with a Java domain, such as org or com, apply to the SAS Micro Analytic Service REST service. In addition, other loggers exist that log information about memory usage and profiling.

Some important SAS Micro Analytic Service REST service loggers include the following:

- **com.sas.mas.service.rest.controllers.ModulesController**: Logs all REST calls to SAS Micro Analytic Service.
- **com.sas.mas.impl.MASFactoryImpl**: Logs environment variables that are presented to SAS Micro Analytic Service core.
- **MAS_MEM.NATIVE**: Logs memory usage by SAS Micro Analytic Service core when executing module code.
- **MAS_MEM.CODE**: Logs memory usage by SAS Micro Analytic Service core when compiling module code.
- **MAS_PROFILE**: Logs sampled timings for compilation and execution of module code.

Logging Levels

The logging levels FATAL, ERROR, WARN, INFO, DEBUG, and TRACE are supported. Consider the following when you are setting logging levels:

- Normal operations, such as start-up and shutdown, are logged at the INFO level. Detailed information is logged at the DEBUG and TRACE level.
- For normal operations, it is recommended that you enable either the ERROR level or the WARN level.
- More verbose and frequent information, such as memory and profile logging, is logged only at the TRACE level.
- Enabling DEBUG and TRACE typically affects performance. Therefore, it is recommended that the DEBUG and TRACE levels are used only during system sizing, performance tuning, or when troubleshooting issues.

SAS Micro Analytic Service Security and Authorization

Access to the REST API endpoints in SAS Micro Analytic Service is determined by authorization rules. For information about how to modify authorization rules, see *SAS Viya Administration: General Authorization*.

Upon installation, authorization rules are created for each SAS Micro Analytic Service endpoint. For each rule, permissions are granted to all authenticated users for specific operations, such as CREATE, READ, UPDATE, and DELETE.

It is recommended that your system administrator perform the following tasks to control access to the SAS Micro Analytic Service, according to your site requirements:

- create new groups of users or update existing groups
- modify the **Principal** value to assign the applicable group
- modify the **Setting** value (if necessary)

For example, the following settings show the default Create and Delete permissions for the SAS Micro Analytic Service:

Object URI	Principal	Setting	Permissions
/microanalyticScore/modules/*	Authenticated Users	Grant	Create, Delete

You can isolate Create and Delete access to the service by creating a new group and assigning that group as the **Principal** value. For example, the following settings show that a new group, called MAS Administrators, is the only group with Create and Delete permissions for the SAS Micro Analytic Service:

Object URI	Principal	Setting	Permissions
/microanalyticScore/modules/*	MAS Administrators	Grant	Create, Delete

For information about creating new user groups, see *SAS Viya Administration: Identity Management*.

Secure DS2 HTTP Package Usage

The DS2 HTTP package supports HTTP and HTTPS endpoints. If the default list of trusted CA certificates does not enable access to all of the secure endpoints that you want to reach, refer to *Encryption in SAS Viya: Data in Motion* for more information. If environment variables such SSLCALISTLOC are needed, they can be added to the **`/opt/sas/viya/config/etc/sysconfig/microanalyticservice.conf`** file.

When an HTTP endpoint requires client authentication, it responds to the client with its list of supported authentication mechanisms. The DS2 HTTP package currently supports two of the three most common authentication mechanisms—Basic and Negotiate. It does not support the Digest mechanism. Because Basic authentication does not provide any credential confidentiality, it should be used only when the data is being encrypted through TLS.

The DS2 HTTP package supports certain security-related methods (for example, setOAuthToken, addSASOAuthToken, setUsername, setPassword, setProxyURL, setProxyUsername, and setProxyPassword). For more information, see“DS2 HTTP Package Methods, Operators, and Statements” in *SAS DS2 Language Reference*.

The Negotiate mechanism supports Kerberos and, when it is used on Windows, NTLM is also supported. For more information, see “Using the HTTP Package” in *SAS DS2 Programmer's Guide*.

Moving Objects by Using the SAS Viya Transfer Service

You can use the SAS Viya transfer service command-line interface (CLI) to move SAS Micro Analytic Service content from one environment to another. To use the CLI, you must be logged in to SAS Viya at the command line. To use the transfer commands, you must be logged in to the source and target environments with an account that has administrator privileges. For more information about the CLI and the transfer service, see SAS Viya Administration: Using the Command-Line Interfaces.

This section contains an example that shows how to transfer modules from a source environment to a target environment. The commands in this example use the named profiles “Source” and “Target”. For specific information about how to use a named

profile, see "Use a Profile to Sign In" in *SAS Viya Administration: Using the Command-Line Interfaces*.

To move content using the transfer service:

1. Create a JSON file of type ExportRequest that includes the IDs of the objects to be exported from the source environment. For example, this sample file contains module IDs exec_module and decision_08c8c3b3_bdb8_4be5_967. The name of the file is export.json.

```
{
   "name": "myTransfer",
   "items":  [
     "/microanalyticScore/modules/exec_module",
     "/microanalyticScore/modules/decision_08c8c3b3_bdb8_4be5_967"
   ]
}
```

2. Create the package file in the source environment:

```
sas-admin --profile Source transfer export --request @export.json
```

   Note: Be sure to note the transfer package ID that is returned by this command.

3. Download the package file from source environment using the transfer package ID obtained in step 2. The following command downloads the package file to the TransferPackage.json file:

```
sas-admin --profile Source transfer download --id transfer-package-ID --file TransferPackage.json
```

4. Upload the package file to the target environment:

```
sas-admin --profile Target transfer upload --file TransferPackage.json
```

   Note: Be sure to note the transfer package ID.

5. Create an import request to promote the transfer package to the target environment using the transfer package ID obtained in step 4:

```
sas-admin --profile Target transfer import --id transfer-package-ID
```

6. (Optional) You can use the following REST API request to confirm that the transferred modules are available:

```
GET service_endpoint/microanalyticScore/modules
```

   Note: service_endpoint represents the service endpoint in the "Target" profile (for example, http://localhost).

Part 4

Appendixes

Appendix 1
Executing Python Modules in DS2 Modules *139*

Appendix 2
SAS Micro Analytic Service Return Codes *149*

Appendix 3
REST Server Error Messages and Resolutions *161*

Appendix 4
Table Service Driver Reference *167*

Appendix 5
SAS Micro Analytic Service Tuning Guidelines *203*

Appendix 6
Applying a New License *205*

Appendix 1

Executing Python Modules in DS2 Modules

DS2 Interface to Python **139**
Overview 139
About Using a PyMAS Package 140

Sample DS2 Module Operations **141**

Configuring Support for a DS2 PyMAS Package **146**
About Using the PyMAS Package 146
Enabling PyMAS Package Support 146

DS2 Interface to Python

Overview

DS2 modules, running in SAS Micro Analytic Service, can publish and execute Python modules. Note that Python must be available for SAS Micro Analytic Service to publish and execute Python modules. For information about the environment variables that are required to enable Python to run in SAS Micro Analytic Service, see "Enabling PyMAS Package Support" on page 146.

DS2 packages that execute outside SAS Micro Analytic Service, such as those within a PROC DS2 program block, can also publish and execute Python modules by using interfaces that are provided via the DS2 PyMAS package.

For information about enabling support for PyMAS, see "Configuring Support for a DS2 PyMAS Package" on page 146.

For information about configuring the environment variables necessary to enable Python to run in SAS Micro Analytic Service, see the resource for your environment:

- SAS Event Stream Processing: Chapter 8, "Python Support in SAS Micro Analytic Service," on page 59
- SAS Intelligent Decisioning: Chapter 13, "Python Support in SAS Micro Analytic Service," on page 111

Note: Python code that is supplied to the PyMAS publish method is not compiled when the DS2 package is published to SAS Micro Analytic Service. Instead, the Python code is compiled when the DS2 module code is executing the PyMAS publish method. The useModule method can be used to specify a Python module that has

been previously published. This enables multiple PyMAS package instances to share the same Python module.

About Using a PyMAS Package

Each PyMAS package instance represents exactly one Python module revision. You can create as many instances as you require, allowing multiple modules to be used.

As is the case when calling any package from DS2, it is recommended that you always check return codes where available, and return any error codes by using an output argument from your DS2 method.

Important: When a DS2 module is executing in SAS Micro Analytic Service, you must conditionally initialize a package-scoped PyMAS package variable and publish the Python module once. Here is an example:

```
package pyscore;
    dcl package pymas py;
    dcl package logger logr('App.tk.MAS');
    dcl varchar(4096) character set utf8 pycode;
    dcl int revision;
    method score( double a, double b,
                  in_out int rc,
                  in_out double c,
                  in_out double d );
        if null(py) then do;
            py = _new_ pymas();
            rc = py.useModule('mypymodule', 1);
            if rc then do;
                rc = py.appendSrcLine('# The first Python function:');
                rc = py.appendSrcLine('def domath1(w, x):');
                rc = py.appendSrcLine('  "Output: y, z"');
                rc = py.appendSrcLine('  y = w * x');
                rc = py.appendSrcLine('  z = w / x');
                rc = py.appendSrcLine('  return y, z');
                pycode = py.getSource();
                revision = py.publish(pycode, 'mypymodule');
                if revision lt 1 then do;
                    logr.log( 'e', 'py.publish() failed.');
                    rc = -1;
                    return;
                end; /* End of if revision less than 1 */
            end; /* End of if useModule failed */
            rc = py.useMethod('domath1');
            if rc then return;
        end; /* End of if null(py)*/
        rc = py.setDouble('w',a);        rc = py.setDouble( 'x', b );
        rc = py.execute();
        c = py.getDouble( 'y' );
        d = py.getDouble( 'z' );
    end;
endpackage;
```

Sample DS2 Module Operations

Here are some operations that a DS2 module would typically perform.

Calling publish() compiles your Python module and sets it as the module that is represented by this PyMAS instance. Subsequent PyMAS function calls, such as setting values and executing methods, operate on this module. The Python code is passed as a string in the first argument. Pass the name that you want to give to your new Python module in the second argument. publish() returns the revision number that SAS Micro Analytic Service assigned to your new module. You could use this revision number later to execute or delete a specific revision of your module. If you do not specify a revision number, the latest revision is assumed. If your Python code fails to publish (because of syntax errors, for example), then -1 is returned for the revision number.

```
revision = py.publish(pgm, moduleName);
```

Rather than publishing a Python module from DS2, you might need to specify a previously published Python module. In this case, you can call useModule() instead of publish(). If a module was already associated with your PyMAS instance before calling useModule(), then useModule() disassociates the current module from the instance before making the specified module current.

```
rc = py.useModule(moduleName, revision);
```

Before calling Python, you must tell the PyMAS instance which method to execute. This is accomplished by calling useMethod(). In addition to specifying the method (Python function) to call, useMethod() also validates that the method exists within the current module, prepares the PyMAS instance to receive the input values for the specific method arguments, and prepares to return any output values from the method execution.

```
rc = py.useMethod(methodName);
```

Call the type-specific setter methods to set input values before executing the method. Because these setters store arguments by name, they can be called in any order, and they insert the values in the correct positions:

```
py.setDouble('airflow', sensor_maf);
```

Since the DS2 package instance represents a single revision, the execute() method needs no arguments.

```
rc = py.execute();
```

After execution, call getters to retrieve the results.

```
score = py.getDouble('credit_score');
```

Scalar argument setters are of the form:

```
return_code = set<type>(name, value)
```

Scalar argument getters are of the form:

```
value = get<type>(name)
```

Array argument setters are of the form:

```
rc = set<type>Array(name, array-value)
```

Array argument getters are of the following form.

Note: DS2 passes arrays and output values by reference.

```
get<type>Array(name, array-value, rc)
```

The example below assumes that you have declared your package as py. The character string variables python_source_code and my_module_name contain the Python source code and the name that will be associated with the published module.

```
dcl package pymas py;
dcl int rc;
dcl bigint result;
py = _new_ pymas();
rc = py.publish(python_source_code, my_module_name);
rc = py.useMethod('func1');
py.setString('inString', 'A string');

py.execute()

bigintVar = py.getLong('outLongVar');
```

The complete set of DS2 package methods follows, where **rc** is the integer return code, and **py** is the package instance.

Methods for Python module management and execution:

```
rc = py.appendSrcLine( python_src_line );
python_source_code = py.getSource();
rc = py.publish(python_source_code, 'module_name');
rc = py.remove();
rc = py.isLoaded(); // returns true is Python is available and false otherwise
revision = py.getRevisionNumber();
rc = py.setTimeZone(time_zone_identifier);
rc = py.execute();
```

Scalar argument setters:

```
rc = py.setString(argument_name, value);
rc = py.setBool(argument_name, value);
rc = py.setLong(argument_name, value);
rc = py.setInt(argument_name, value);
rc = py.setDouble(argument_name, value);
rc = py.setDateTime(argument_name, value);
rc = py.setDate(argument_name, value);
rc = py.setTime(argument_name, value);
```

Scalar argument getters:

```
string_value      = py.getString(argument_name);
int_value         = py.getBool(argument_name);
long_value        = py.getLong(argument_name);
int_value         = py.getInt(argument_name);
double_value      = py.getDouble(argument_name);
date_time_value   = py.getDateTime(argument_name);
date_value        = py.getDate(argument_name);
time_value        = py.getTime(argument_name);
```

Array argument setters:

```
rc = py.setStringArray(argument_name, string_array);
rc = py.setBoolArray(argument_name, integer_array);
rc = py.setLongArray(argument_name, bigint_array);
rc = py.setIntArray(argument_name, integer_array);
rc = py.setDoubleArray(argument_name, double_array);
```

```
rc = py.setDateTimeArray(argument_name, date_time_array);
rc = py.setDateArray(argument_name, date_array);
rc = py.setTimeArray(argument_name, time_array);
```

Array argument getters:

```
py.getStringArray(argument_name, string_array, rc);
py.getBoolArray(argument_name, integer_array, rc);
py.getLongArray(argument_name, bigint_array, rc);
py.getIntArray(argument_name, integer_array, rc);
py.getDoubleArray(argument_name, double_array, rc);
py.getDateTimeArray(argument_name, date_time_array, rc);
py.getDateArray(argument_name, date_array, rc);
py.getTimeArray(argument_name, time_array, rc);
```

Note the following important information about Python and SAS Micro Analytic Service:

- For Python 2.*x*, SAS Micro Analytic Service supports only the use of ASCII characters.
- For Python 3.*x*, if SAS Micro Analytic Service attempts to publish a Python module that includes syntax to define the source code encoding, the encoding must be UTF-8. This type of encoding is known as a Python *magic comment*.
- The PyMAS package method getters do not support converting the value that is returned from Python to the type named in the getter method. The value of the output argument that is returned from Python must be consistent with the PyMAS getter method that is called. For example, consider the getDouble method example:

  ```
  myintvar = py.getDouble('pyoutarg1');
  ```

  The **pyoutarg1** output argument that is returned from Python function must be of type double, but the assignment statement can coerce the value to an integer.

  Although Python packages might support abstract types that provide an additional layer on top of the built-in scalar types, such as integer and float, they cannot be returned from SAS Micro Analytic Service public functions. Packages that have these abstract types provide functions that can be used to extract the built-in type. Here is an example:

  ```
  return numpy.float(x), numpy.int(y)
  ```

- Python modules can use other Python modules that are published to SAS Micro Analytic Service. DS2 PyMAS package instances can use the useModule method to specify a previously published module, instead of publishing a revised Python module.

When using PROC DS2 in a SAS session to create a PyMAS package instance, you cannot provide the Python program as one quoted literal string. The reason is that the SAS tokenizer strips out the embedded line-ending characters, causing indention problems in the Python code. In this situation, a PyMAS package's appendSrcLine() and getSource() methods can be used. When used together, these two methods produce a DS2 character variable containing the lines of code concatenated together with embedded linefeed characters separating the lines of Python code. Once you have added each line of your Python code to the PyMAS package instance using the appendSrcLine() method, then use the getSource() method to retrieve the complete program into a DS2 character variable. The variable can then be provided as the first input argument to the PyMAS publish() method. An error is logged when you use the PyMAS useModule method to specifya module that has not been published. The following PROC DS2 example encounters the error on the first execution of the score

method. The score method checks the return code and publishes the module. Subsequent executions of the score method do not generate the error.

```
%macro chkrc; if rc then put rc=; %mend;
%macro addln( line ); rc = pm.appendSrcLine( &line ); %chkrc; %mend;

/* Input data for the test.*/
data tstinput; a = 8;  b = 4; output; a = 10; b = 2; output;
run;

proc ds2;
    ds2_options sas;
    package mypkg;
        dcl package pymas pm;
        dcl package logger logr('App.tk.MAS');
        dcl varchar(67108864) character set utf8 pycode;
        dcl int revision;

        method usefunc( varchar(256) pyfuncname, in_out int rc );
            rc = pm.useMethod( pyfuncname );
            if rc then logr.log( 'E', 'pm.useMethod() failed.');
        end;

        method score( double a, double b,
                      in_out int rc,
                      in_out double c, in_out double d );
            if null(pm) then do;
                pm = _new_ pymas();
                rc = pm.useModule( 'mypymodule', 1 );
                if rc then do;
                    %addln( '# The first Python function:' )
                    %addln( 'def domath1(a, b):' )
                    %addln( '  "Output: c, d"' )
                    %addln( '  c = a * b' )
                    %addln( '  d = a / b' )
                    %addln( '  return c, d' )
                    %addln( '' )
                    %addln( '# Here is the second function:' )
                    %addln( 'def domath2(a, b):' )
                    %addln( '  "Output: c, d"' )
                    %addln( '  c,d = domath1( a, b )' )
                    %addln( '  return c, d' )
                    if rc then do;
                        logr.log( 'E', 'pm.appendSrcLine() failed.');
                        pm = null;
                        return;
                    end;
                    pycode = pm.getSource();
                    revision = pm.publish( pycode, 'mypymodule' );
                    if ( revision < 1 ) then do;
                        logr.log( 'E', 'pm.publish() failed.');
                        rc = -1;
                        pm = null;
                        return;
                    end;
                end;
```

```
                rc = pm.useMethod( 'domath1' );
                if rc then do;
                        logr.log( 'E', 'pm.useMethod() failed.');
                        return;
                end;
            end;
            rc = pm.setDouble( 'a', a );   if rc then return;
            rc = pm.setDouble( 'b', b );   if rc then return;
            rc = pm.execute();             if rc then return;
            c = pm.getDouble( 'c' );
            d = pm.getDouble( 'd' );
        end;
    endpackage;

    data _null_;
        dcl package logger logr();
        dcl package mypkg t();
        dcl int rc;
        dcl double a b c d;

        method run();
            rc = 0;
            c = d = .;
            set tstinput;
            t.score( a, b, rc, c, d );
            if rc then do;
                logr.log( 'E', 'rc=$s', rc );
                stop;
            end;
            logr.log( 'I', 'Results: a=$s   b=$s   c=$s   d=$s',
                      a, b, c, d );
        end;

        method term();
            if not rc then do;
                t.usefunc( 'domath2', rc );
                if rc then do;
                    logr.log( 'E', 'rc=$s', rc );
                    return;
                end;
                a = 6; b = 3;
                t.score( a, b, rc, c, d );
                if rc then
                    logr.log( 'E', 'rc=$s', rc );
                else
                    logr.log( 'I', 'Results: a=$s   b=$s   c=$s   d=$s',
                              a, b, c, d );
            end;
        end;
    enddata;
    run;
quit;
```

Configuring Support for a DS2 PyMAS Package

About Using the PyMAS Package

Here are some examples of how a PyMAS package might be used:

- In SAS Intelligent Decisioning, you have DS2 code that uses a PyMAS package that is executed using the microanalyticScore microservice.
- In SAS Model Manager, you have DS2 code that uses a PyMAS package that is executed using the Compute server or the CAS server.
- In SAS Studio, you use PROC DS2, and the PROC uses a PyMAS package that is executed using the Workspace server.

Enabling PyMAS Package Support

To enable support of a PyMAS package in environments that execute DS2 packages, you must add Python environment configuration commands to the appropriate scripts that are used by those services or servers during initialization.

These are the SAS Micro Analytic Service environment variables that must be set, with examples for both UNIX and Windows platforms:

- MAS_M2PATH: Specifies the absolute path to the mas2py.py file. This file is included with SAS Micro Analytic Service. It is used to execute Python code within a Python process that is launched by SAS Micro Analytic Service. Here are some examples:
  - UNIX platform:

    ```
    MAS_M2PATH=/opt/sas/viya/home/SASFoundation/misc/embscoreeng/mas2py.py
    export MAS_M2PATH
    ```

    or

    ```
    MAS_M2PATH=/opt/sas/spre/home/SASFoundation/misc/embscoreeng/mas2py.py
    export MAS_M2PATH
    ```

  - Windows platform:

    ```
    set MAS_M2PATH=C:\Program Files\SAS\VIYA\SASFoundation\misc\
    embscoreeng\mas2py.py
    ```

    or

    ```
    set MAS_M2PATH=C:\Program Files\SAS\SPRE\SASFoundation\misc\
    embscoreeng\mas2py.py
    ```

- MAS_PYPATH: Indicates the absolute path to the Python executable. Here are some examples:
  - UNIX platform:

    ```
    MAS_PYPATH=/usr/bin/python
    export MAS_PYPATH
    ```

  - Windows platform:

    ```
    set MAS_PYPATH=c:\python\python.exe
    ```

The scripts are as follows:

- microanalyticScore microservice: **/opt/sas/viya/config/etc/sysconfig/microanalyticservice.conf**

  Note: If this file does not exist, you must create it.

- Compute server: **/opt/sas/viya/config/etc/sysconfig/compsrv/default/sas-compsrv**
- Workspace server: **/opt/sas/viya/config/etc/workspaceserver/default/workspaceserver_usermods.sh**
- SAS Cloud Analytic Services (CAS server): **/opt/sas/viya/config/etc/cas/default/cas_usermods.settings**

  Note: The user's identity must exist in the CASHostAccountRequired group. For information, see "The CASHostAccountRequired Custom Group" in *SAS Viya Administration: Identity Management*.

Appendix 2

SAS Micro Analytic Service Return Codes

The SAS Micro Analytic Service core component, tkmas, supports the following return codes. Depending on logging settings, an associated message might be logged. When a message is logged, any substitution parameters (indicated by %s for string and %d for number) are filled in. The other SAS Micro Analytic Service interface layers, such as the Java interface and the REST interface, might log additional messages that are not listed below.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958744063	0x8b3ff001U	MASBadArgs	Invalid arguments.
-1958744062	0x8b3ff002U	MASInternalError	Internal error.
-1958744061	0x8b3ff003U	MASFailure	SAS Micro Analytic Service encountered a failure.
-1958744060	0x8b3ff004U	MASFail	%s encountered a failure.
-1958744059	0x8b3ff005U	MASUnexFail	%s encountered an unexpected failure.
-1958744058	0x8b3ff006U	MASUnexInternal	%s encountered an unexpected internal failure.
-1958744057	0x8b3ff007U	MASUnexFailIn	%s encountered an unexpected failure in %s.
-1958744056	0x8b3ff008U	MASFailIn	%s encountered a failure in %s.
-1958744055	0x8b3ff009U	MASFailWithText	%s encountered a failure in %s: %s.
-1958744054	0x8b3ff00aU	MASSFGCBLock	Failed to obtain the SFGCB lock.
-1958744053	0x8b3ff00bU	MASExeLock	Failed to obtain the .exe lock.
-1958744052	0x8b3ff00cU	MASLockCreate	Failed to create the %s lock.
-1958744051	0x8b3ff00dU	MASEventCreate	Failed to create the %s event for thread %d.
-1958744050	0x8b3ff00eU	MASThreadCreate	Failed to create SAS Micro Analytic Service worker thread %d of %d.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958744049	0x8b3ff00fU	MASCPUCount	Failed to determine the number of CPUs. Setting the number of worker threads to %d.
-1958744048	0x8b3ff010U	MASThreadCount	The number of threads requested, %d, exceeds the limit. The maximum allowable threads = %d times the number of CPUs = %d.
-1958744047	0x8b3ff011U	MASThreadPoolSize	Worker thread pool size is set to: %d.
-1958744046	0x8b3ff012U	MASInitAlready	SAS Micro Analytic Service was already initialized.
-1958744045	0x8b3ff013U	MASInitFailed	SAS Micro Analytic Service failed to initialize.
-1958744044	0x8b3ff014U	MASNotLicensed	SAS Micro Analytic Service is not licensed.
-1958744043	0x8b3ff015U	MASLicSvcInitFailed	License service failed to initialize.
-1958744042	0x8b3ff016U	MASNotInitialized	SAS Micro Analytic Service is not initialized.
-1958744041	0x8b3ff017U	MASTermFailed	SAS Micro Analytic Service failed to terminate successfully.
-1958744040	0x8b3ff018U	MASArgTrunc	The maximum size of parameter %d in the %s call is not large enough, and the value has been truncated at %d characters.
-1958744039	0x8b3ff019U	MASCompStatus	Compiler encountered status 0x%X.
-1958744038	0x8b3ff01aU	MASUnsupportedType	Unsupported type.
-1958744037	0x8b3ff01bU	MASUnknownType	Unknown type.
-1958744036	0x8b3ff01cU	MASNoSuchPackage	Package not found.
-1958744035	0x8b3ff01dU	MASNoSuchMethod	Method not found.
-1958744034	0x8b3ff01eU	MASNoSuchRevision	Revision not found.
-1958744033	0x8b3ff01fU	MASRevisionGet	Failed to get revision.
-1958744032	0x8b3ff020U	MASNoSuchModule	Module not found.
-1958744031	0x8b3ff021U	MASNoSuchUserContext	User context not found.
-1958744030	0x8b3ff022U	MASModuleCtxtCreate	Failed to create module context.
-1958744029	0x8b3ff023U	MASUserCtxtCreate	Failed to create user context.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958744028	0x8b3ff024U	MASArgTypeMismatch	Argument type mismatch.
-1958744027	0x8b3ff025U	MASArgCoutMismatch	Argument count mismatch.
-1958744026	0x8b3ff026U	MASClientCodegenError	Code generation error.
-1958744025	0x8b3ff027U	MASDS2CompileError	DS2 compilation error.
-1958744024	0x8b3ff028U	MASDS2RuntimeError	DS2 run-time error.
-1958744023	0x8b3ff029U	MASTKGNoEntryPoint	Code generation did not find an entry point.
-1958744022	0x8b3ff02aU	MASTKGGenericError	Code generation generic error.
-1958744021	0x8b3ff02bU	MASInvalidRequest	Invalid request.
-1958744020	0x8b3ff02cU	MASMissingEntryPoints	Missing entry points.
-1958744019	0x8b3ff02dU	MASUnassignedInput	Unassigned input.
-1958744018	0x8b3ff02eU	MASInternalOnly	Internal only.
-1958744017	0x8b3ff02fU	MASOnlyValidForDS2	Valid only for DS2 code.
-1958744016	0x8b3ff030U	MASOnlyValidForC	Valid only for C code.
-1958744015	0x8b3ff031U	MASExecutionException	Exception occurred during execution.
-1958744014	0x8b3ff032U	MASCompilationException	Exception occurred during compilation.
-1958744013	0x8b3ff033U	MASDS2ThreadUnsupported	DS2 thread unsupported.
-1958744012	0x8b3ff034U	MASTKEDSError	DS2 error.
-1958744011	0x8b3ff035U	MASUnrecognizedLanguage	Unrecognized language.
-1958744010	0x8b3ff036U	MASUnspecifiedDataType	Unspecified data type.
-1958744009	0x8b3ff037U	MASTKThreadingError	Threading error.
-1958744008	0x8b3ff038U	MASFatalProgRepoLost	Program repository lost.
-1958744007	0x8b3ff039U	MASSaveToRepo	Failed to save to repository.
-1958744006	0x8b3ff03aU	MASLog4SASCfgFailed	Logging configuration failed.
-1958744005	0x8b3ff03bU	MASDS2CompileStart	User context '%s' compiling module '%s' on thread %d.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958744004	0x8b3ff03cU	MASDS2CompileFinish	User context '%s' module '%s' thread %d compilation succeeded.
-1958744003	0x8b3ff03dU	MASDS2CompileFailed	User context '%s' module '%s' thread %d new revision failed, RC = %d.
-1958744002	0x8b3ff03eU	MASStartup	*** SAS Micro Analytic Service Started ***
-1958744001	0x8b3ff03fU	MASShutdown	service Micro Analytic Score Service shutdown
-1958744000	0x8b3ff040U	MASAsyncException	SAS Micro Analytic Service received async exception code %d.
-1958743999	0x8b3ff041U	MASAsyncInitFailed	SAS Micro Analytic Service failed to install async exception handler.
-1958743998	0x8b3ff042U	MASShutdownJNI	SAS Micro Analytic Service calling JVM System.exit(0).
-1958743997	0x8b3ff043U	MASExecDeletePending	Attempt to execute method %s while deletion pending for module context %s revision %d.
-1958743996	0x8b3ff044U	MASMTXDeletePending	Attempt to add module context &s while deletion pending for user context %s.
-1958743995	0x8b3ff045U	MASRevDeletePending	Attempt to create revision while deletion pending for module context %s.
-1958743994	0x8b3ff046U	MASRevDelDeletePending	Attempt to delete revision while deletion pending for module context %s.
-1958743993	0x8b3ff047U	MASRevDelRefCount	Pending delete called for module context %s with ref count %d.
-1958743992	0x8b3ff048U	MASRevDelRefCountError	Delete called for module context %s with ref count %d.
-1958743991	0x8b3ff049U	MASMTXDelete	Garbage collection is deleting module context %s.
-1958743990	0x8b3ff04aU	MASCTXDeletePending	Attempt to delete user context %s while being deleted by another thread.
-1958743989	0x8b3ff04bU	MASCTXGetCDTDelPending	Attempt to retrieve creation time from user context %s while deletion pending.
-1958743988	0x8b3ff04cU	MASCTXGetMDTDelPending	Attempt to retrieve modified time from user context %s while deletion pending.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743987	0x8b3ff04dU	MASMTXGetCDTDelPending	Attempt to retrieve creation time from module context %s while deletion pending.
-1958743986	0x8b3ff04eU	MASMTXGetMDTDelPending	Attempt to retrieve modified time from module context %s while deletion pending.
-1958743985	0x8b3ff04fU	MASMTXGetRevDelPending	Attempt to retrieve highest revision from module context %s while deletion pending.
-1958743984	0x8b3ff050U	MASMTXGetIUODelPending	Attempt to retrieve internal use flag from module context %s while deletion pending.
-1958743983	0x8b3ff051U	MASRevGetCDTDelPending	Attempt to retrieve revision %d creation time from module context %s while deletion pending.
-1958743982	0x8b3ff052U	MASMTXGetMsgDelPending	Attempt to retrieve compilation messages from module context %s while deletion pending.
-1958743981	0x8b3ff053U	MASMTXRegDeletePending	Attempt to register name while deletion pending for module context %s.
-1958743980	0x8b3ff054U	MASMTXLangDelPending	Attempt to retrieve language of module context %s while deletion pending.
-1958743979	0x8b3ff055U	MASMTXGetDispDelPending	Attempt to retrieve display name from module context %s while deletion pending.
-1958743978	0x8b3ff056U	MASMTXGetCSrcDelPending	Attempt to retrieve C source code from module context %s revision %d while deletion pending.
-1958743977	0x8b3ff057U	MASCTXGetPkgsDelPending	Attempt to retrieve packages from user context %s while deletion pending.
-1958743976	0x8b3ff058U	MASMTXGetMthsDelPending	Attempt to retrieve methods from module context %s while deletion pending.
-1958743975	0x8b3ff059U	MASNoSuchEntryPoint	Entry point not found.
-1958743974	0x8b3ff05aU	MASMTXGetSigDelPending	Attempt to retrieve method %s signature from module context %s while deletion pending.
-1958743973	0x8b3ff05bU	MASCTXLdOOTBDelPending	Private load out-of-the-box packages for user context %s while deletion pending.
-1958743972	0x8b3ff05cU	MASCTXRegIntDelPending	Attempt to publish internal package %s to user context %s while deletion pending.
-1958743971	0x8b3ff05dU	MASCTXRemIntDelPending	Attempt to remove internal package %s from user context %s while deletion pending.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743970	0x8b3ff05eU	MASCreateGCAFailed	Attempt to create garbage collection control structures failed.
-1958743969	0x8b3ff05fU	MASGarbageCollection	Garbage collection interval.
-1958743968	0x8b3ff060U	MASGarbageCollectionDel	Garbage collection found assets ready to delete.
-1958743967	0x8b3ff061U	MASGCException	Exception occurred during garbage collection run.
-1958743966	0x8b3ff062U	MASProgRepoUpdateError	Error obtaining exclusive lock to update DS2 program repository.
-1958743965	0x8b3ff063U	MASCTXDelete	Garbage collection is deleting user context %s.
-1958743964	0x8b3ff064U	MASRevDelete	Garbage collection is deleting module context %s revision %d.
-1958743963	0x8b3ff065U	MASDS2Fatal	Module context %s revision %d generated fatal run-time exception. Deleting revision.
-1958743962	0x8b3ff066U	MASGarbageCollectionTerm	Garbage collection is freeing control assets during shut down.
-1958743961	0x8b3ff067U	MASShutdownHang	Worker thread did not interrupt after %d seconds during shutdown.
-1958743960	0x8b3ff068U	MASGCInvalidIntervalHigh	Specifies that the garbage collection interval is above the maximum. Setting to default value.
-1958743959	0x8b3ff069U	MASGCInvalidIntervalLow	Specifies that the garbage collection interval is below the minimum. Setting to default value.
-1958743958	0x8b3ff06aU	MASGCInvalidGraceHigh	Specifies that the grace period is above the maximum. Setting to default value.
-1958743957	0x8b3ff06bU	MASGCInvalidGraceLow	Specifies that the grace period is below the minimum. Setting to default value.
-1958743956	0x8b3ff06cU	MASGCMissingInterval	Garbage collection interval is not specified. Setting to default value.
-1958743955	0x8b3ff06dU	MASGCMissingGracePeriod	Grace period is not specified. Setting to default value.
-1958743954	0x8b3ff06eU	MASModuleStats	Check the log for module statistics.
-1958743953	0x8b3ff06fU	MASInvalidDS2Connection	Attempt to create TKTS driver connection failed.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743952	0x8b3ff070U	MASDS2FatalRecompiled	DS2 package fatal error. Auto-recompile succeeded.
-1958743951	0x8b3ff071U	MASDS2FatalRecompFailed	DS2 package fatal error. Transaction failed. Recompile failed. Ejecting revision.
-1958743950	0x8b3ff072U	MASDS2RevisionEjected	DS2 package fatal error. Max retry exceeded. Ejecting revision. Correct and republish.
-1958743949	0x8b3ff073U	MASDBConnLost	Connection to the database lost. Check the log for details.
-1958743948	0x8b3ff074U	MASDBConnReestablished	Lost connection reestablished for user context.
-1958743947	0x8b3ff075U	MASDBConnRetryLimit	Maximum connection retry attempts exceeded for user context.
-1958743946	0x8b3ff076U	MASDBConnDoesNotExist	Attempt to execute SQLSTMT, when no connection exists.
-1958743945	0x8b3ff077U	MASDBConnRetryThreadErr	Error while creating database connection retry thread.
-1958743944	0x8b3ff078U	MASDBConnRetryAttempt	Connection retry attempt unsuccessful.
-1958743943	0x8b3ff079U	MASNameRegisterFailed	Unable to register tkmas in the threaded kernel named registry. DS2 programs that call Python scripts will not function.
-1958743942	0x8b3ff07aU	MASDS2PythonNameRequired	AS DS2 Python constructor missing Python module name.
-1958743941	0x8b3ff07bU	MASDS2PythonCreateError	Unable to create SAS Micro Analytic Service DS2 Python package.
-1958743940	0x8b3ff07cU	MASDS2PythonInitError	Unable to initialize support for SAS Micro Analytic Service DS2 Python package.
-1958743939	0x8b3ff07dU	MASUnsupportedFunction	Unsupported function.
-1958743938	0x8b3ff07eU	MASDS2NotInitialized	Attempt to perform action on uninitialized SAS Micro Analytic Service DS2 Python package.
-1958743937	0x8b3ff07fU	MASDS2PythonParmError	SAS Micro Analytic Service DS2 Python package parameter mismatch.
-1958743936	0x8b3ff080U	MASDS2PythonArgNameReqd	SAS Micro Analytic Service DS2 Python missing argument name.
-1958743935	0x8b3ff081U	MASDS2PythonArgValueReqd	AS DS2 Python missing argument value.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743934	0x8b3ff082U	MASDS2PythonArgInvalid	SAS Micro Analytic Service DS2 Python invalid argument value.
-1958743933	0x8b3ff083U	MASDS2PythonThreadError	Invalid operation: DS2 callback into SAS Micro Analytic Service received an unrecognized thread.
-1958743932	0x8b3ff084U	MASPythonCompileEx	Exception thrown while initializing Python or compiling Python script.
-1958743931	0x8b3ff085U	MASDS2InvalidMaxRecomp	Invalid maximum DS2 recompile count given. Setting to default value.
-1958743930	0x8b3ff086U	MASDBInvalidIntervalHigh	Specified DBMS connection retry interval is above the maximum. Setting to default value.
-1958743929	0x8b3ff087U	MASDBInvalidIntervalLow	Specified DBMS connection retry interval is below the minimum. Setting to default value.
-1958743928	0x8b3ff088U	MASDBInvalidMaxRetry	Invalid setting for maximum DBMS reconnection attempts. Setting to default value.
-1958743927	0x8b3ff089U	MASDBCreateConnErr	SAS Micro Analytic Service failed to create a connection.
-1958743926	0x8b3ff08aU	MASDBCreateConn	SAS Micro Analytic Service created a connection.
-1958743925	0x8b3ff08bU	MASGCCanBeDeleted	Garbage collection is checking module context for deletion pending.
-1958743924	0x8b3ff08cU	MASRepoLockRemovePriv	Locking program repository to remove internal package.
-1958743923	0x8b3ff08dU	MASRepoUnlockRemovePriv	Released program repository lock after removing internal package.
-1958743922	0x8b3ff08eU	MASRepoLockRemoveRev	Locking program repository to remove module context.
-1958743921	0x8b3ff08fU	MASRepoUnlockRemoveRev	Released program repository lock, after removing module context.
-1958743920	0x8b3ff090U	MASRepoLockCreate	Creating a lock for user context.
-1958743919	0x8b3ff091U	MASRepoLockDestroy	Destroying a lock for user context.
-1958743918	0x8b3ff092U	MASRepoLockPackageComp	Locking program repository during compilation of package.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743917	0x8b3ff093U	MASRepoUnlockPackageComp	Released program repository lock after compilation of package.
-1958743916	0x8b3ff094U	MASRepoUnlockCompCrash	Released program repository lock due to DS2 compiler crash while compiling package.
-1958743915	0x8b3ff095U	MASRepoLockPackageSave	Locking program repository to save package after successful compilation.
-1958743914	0x8b3ff096U	MASRepoUnlockPackageSave	Released program repository after saving package.
-1958743913	0x8b3ff097U	MASRepoLockPackagePriv	Locking program repository to save internal package.
-1958743912	0x8b3ff098U	MASRepoUnlockPackagePriv	Released program repository after saving internal package.
-1958743911	0x8b3ff099U	MASPythonNotLoaded	Python extension not loaded. Python must be installed in order to execute Python within SAS Micro Analytic Service.
-1958743910	0x8b3ff09aU	MASTKTSConnHndlFail	Failed to create a table services connection handle.
-1958743909	0x8b3ff09bU	MASDBDisconnected	SAS Micro Analytic Service disconnected database from user context.
-1958743908	0x8b3ff09cU	MASDBDisconnect	SAS Micro Analytic Service encountered a failure when attempting to disconnect the database from the user context.
-1958743907	0x8b3ff09dU	MASPercentS	Internal error. Check the SAS Micro Analytic Service Core log.
-1958743906	0x8b3ff09eU	MASPythonCompileErr	Error compiling the Python script for the module.
-1958743905	0x8b3ff09fU	MASDS2MissingArray	A missing array argument is not supported with DS2.
-1958743904	0x8b3ff0a0U	MASDS2EmptyArray	An empty array argument is not supported with DS2.
-1958743903	0x8b3ff0a1U	MASDS2ArrayReplaced	Missing or insufficiently sized DS2 array argument has been replaced with new array of size %d.
-1958743902	0x8b3ff0a2U	MASDS2OutputTransError	Error %d when converting CHAR string of length %d to TKChar string.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
-1958743901	0x8b3ff0a3U	MASDS2InputTransError	Error %d when converting TKChar string of length %d to CHAR string.
-1958743900	0x8b3ff0a4U	MASDS2PythonOutputTrans	Error %d when converting Python CHAR string of length %d to TKChar string.
-1958743899	0x8b3ff0a5U	MASDS2PythonInputTrans	Error %d when converting TKChar string of length %d to CHAR string for Python.
-1958743898	0x8b3ff0a6U	MASDBCr8ConnNoSub	SAS Micro Analytic Service created a default data source connection.
-1958743897	0x8b3ff0a7U	MASDBCr8ConnErrNoSub	SAS Micro Analytic Service failed to create a default data source connection.
1958743896	0x8b3ff0a8U	MASDBDisconnNoSub	SAS Micro Analytic Service disconnected from the default data source.
1958743895	0x8b3ff0a9U	MASDBDisconnErrNoSub	SAS Micro Analytic Service encountered a failure when attempting to disconnect from the default data source.
1958743894	0x8b3ff0aaU	MASDS2ScanError	Out of memory or malformed DS2 encountered while scanning the package %s source code prior to dictionary generation.
1958743893	0x8b3ff0abU	MASDS2ParseError	Out of memory or malformed DS2 encountered while parsing the package %s method %s during dictionary generation.
1958743892	0x8b3ff0acU	MASMTXGetDictDelPending	Attempt to retrieve the dictionary from module context %s revision %d while deletion pending.
1958743892	0x8b3ff0adU	MASCFuncProtoNotSupp	Part of the C function prototype is not supported.
1958743890	0x8b3ff0aeU	MASDupModuleName	Module name %s already exists. Module name must be unique within the user context.
1958743889	0x8b3ff0afU	MASDupDS2Package	The DS2 package name %s is already bound to module %s. Separate modules cannot represent the same DS2 package.
1958743888	0x8b3ff0b0U	MASIndexOutOfRangeSet	The index is out of range while setting an argument. Argument %d specified when number of arguments is %d.
1958743887	0x8b3ff0b1U	MASIndexOutOfRangeGet	The index is out of range while retrieving an argument. Argument %d specified when number of arguments is %d.

Return Code	Hexadecimal Code	#define Symbol	Message or Description
1958743886	0x8b3ff0b2U	MASIntTypeExpected	The argument %d in method %Us should be an integral type used to specify the length of the previous argument, which is an array.
1958743885	0x8b3ff0b3U	MASOutArgExpected	The argument %d in method %Us should be an output argument. All input arguments must precede output arguments.
1958743884	0x8b3ff0b4U	MASDS2pymas	DS2 PyMAS package encountered a failure.
1958743883	0x8b3ff0b5U	MASDS2pymasFailIn	DS2 PyMAS package encountered a failure in %Us.
1958743882	0x8b3ff0b6U	MASDS2pymasPubUTF8	DS2 PyMAS package failed to publish module %Us.
1958743881	0x8b3ff0b7U	MASDS2pymasPubTK	DS2 PyMAS package failed to publish module %s.
1958743880	0x8b3ff0b8U	MASDS2pymasUsed	The DS2 PyMAS package's use method has already been called on this package instance. Create a separate PyMAS instances for each method that is used.
1958743879	0x8b3ff0b9U	MASThrdPoolSizeDiff	SAS Micro Analytic Service has already been initialized with a worker thread pool size of %d.
1958743878	0x8b3ff0baU	MASSymbolTableCreateFailed	SAS Micro Analytic Service failed to create a symbol table.
1958743877	0x8b3ff0bbU	MASMethodExecutionFailed	SAS Micro Analytic Service failed to execute a method.

Appendix 3

REST Server Error Messages and Resolutions

The following table contains SAS Micro Analytic Service REST server error messages, as well as possible causes and remedies.

HTTP Code	Error Message	Explanation	Remedy
Errors related to missing resources			
400	The module **`module_ID`** not found.	A module with the specified ID does not exist. This typically occurs when attempting to do one of the following for the specified module: • view, update, delete the module • execute a step of the module • validate data input for the execution of a step of the module	Specify a module ID that exists in the system.
404	The step **`step_ID`** of module **`module_ID`** was not found.	The specified module exists, but the referenced step was not found. This typically occurs when attempting to do one of the following: • access the specified module to execute a step • validate data input for the execution of a step	Specify a step ID that exists in the specified module ID.
404	The submodule **`submodule_ID`** of module **`module_ID`** was not found.	The specified module exists, but the referenced submodule was not found. This can occur when attempting to access the specified submodule to view its properties or source code.	Specify a submodule ID that exists in the specified module ID.
Errors related to create and update operations			
400	Cannot create or update the module. The specified media type **`type_name`** is not supported. The only supported media type is **`type_name`**.	The media type must be text/vnd.sas.source.ds2 or text/application.source.ds2.	Specify the correct media type.

HTTP Code	Error Message	Explanation	Remedy
400	Cannot create or update the module. DS2 and COMPOSITE are the only valid values for language.	The programming language is not DS2 or COMPOSITE.	Supply DS2 or COMPOSITE modules only. Other languages might be supported in the future.
400	Cannot create or update module. The specified module ID must not be empty or missing.	Under certain circumstances, it is possible to specify the module ID as part of the module definition. However, when doing so the module specification must not be an empty string or missing.	Specify a valid module ID string.
400	Cannot create or update the module. Unable to parse the package name.	The DS2 package must be properly formed so that the package name can be parsed.	Supply a properly formed DS2 package.
400	Cannot create or update module. The value of field **field_name** must not be empty or missing.	When creating or updating a module, the payload contains certain mandatory fields, for example, scope, type, and code. Mandatory fields cannot be missing and the associated content must not be null or an empty string.	Assign a valid value for the fields that are referenced in the message.
400	Cannot create or update the module. At least one of the fields **field_name_1** or **field_name_2** must be present.	Either **field_name_1** or **field_name_2** is required input to create or update the module.	Supply either **field_name_1** or **field_name_2**.
400	Cannot create or update the module. Only one of the fields **field_name** or **field_name** can be present.	Only one field name (either **field_name_1** or **field_name_2**) can be present to create or update the module.	Supply either **field_name_1** or **field_name_2**.
400	Cannot create or update module. The field **field_name** must not be repeated.	When creating or updating a module, the payload can contain multiple name value pairs called properties. Property names must be unique.	Ensure that property names are unique.
400	Cannot create or update module. An unexpected end of source was encountered while parsing a comment.	DS2 code that contains comments must be properly delimited.	Refer to *SAS DS2 Language Reference* for information about comment syntax rules.
400	Cannot create or update module due to unmatched single quotation marks.	If strings are defined in the DS2 program, they must be enclosed in single quotation marks.	Refer to *SAS DS2 Language Reference* for information about character constant syntax rules.

HTTP Code	Error Message	Explanation	Remedy
400	Cannot create or update module due to unmatched double quotation marks.	If delimited variables names are defined in the DS2 program, they must be enclosed in matching double quotation marks.	Refer to *SAS DS2 Language Reference* for information about delimited variable reference syntax rules.
400	Cannot create or update module. No DS2 package was found in the source code.	DS2 source was not found.	Refer to *SAS DS2 Language Reference* for information about DS2 package syntax rules.
400	Cannot create or update module. More than one DS2 package found in the source code.	Only a single DS2 package is accepted to create a module.	Ensure that only a single DS2 package is contained in the source code. Multiple packages can be separated and created as separate modules.
400	Cannot create or update module **`module_ID`** due to the following compilation errors: **`error_messages`**.	The DS2 source code contains compilation errors.	Resolve the issues, and then resubmit the source code. For more information, see *SAS DS2 Language Reference*.
400	Cannot create module. The specified module ID **`module_ID`** is already in use.	When creating a module, the specified module ID must not already exist in the system.	Depending on the configuration, the module ID is either supplied or derived from the package name. To avoid this error, specify the appropriate module ID or DS2 package.
400	The module **`module_ID`** is not a COMPOSITE module. Only COMPOSITE modules can have submodules.	This is typically returned when the submodules of a module that does not support submodules are accessed.	Select a COMPOSITE module.
400	Cannot create or update the module. The file **`file_name`** is not accessible from this server.	This is typically returned when the file containing the analytic store module is referenced and the server cannot access the file.	Ensure that the file **`file_name`** exists and is accessible by the web service.
400	Cannot create or update the module. The submodule name **`name`** must not be repeated.	The supplied module definition contains more than one submodule with the same name.	Ensure that the submodule names in the module definition are unique.

HTTP Code	Error Message	Explanation	Remedy
400	Cannot update the module **module_ID**. The language cannot be changed.	The module definition used for updating a module contains a different language value than the original module. After a module is created, the language cannot be changed.	To preserve the same language, supply the same language value or remove the module language field from the module definition. To create a module with a different language, delete the original module and create a new one.
Errors related to accessing or executing the steps of a module, or validating the data needed before execution.			
400	Cannot execute step **step_ID** of module **module_ID**.	The input data supplied to execute a module step is incorrect. Subsequent error messages will provide further details.	Refer to the subsequent messages related to this issue for more information.
400	Expected **integer_value**, input parameters, but received **integer_value**.	An incorrect number of parameters were supplied to execute a module step.	Supply the correct number of parameters.
400	The parameter **parameter_name** is not defined.	The named parameter is not defined for the module step.	Select the correct parameter name or execute a different module step.
400	Cannot assign the value **specified_value** to the parameter **parameter_name** of type **type**.	An inappropriate value was received for the parameter, for example, supplying a string value to an integer parameter.	Specify appropriate matching values.
400	Cannot assign an array of size **integer_value** to the array parameter **parameter_name**. A maximum of **integer_value** elements can be accepted.	The supplied array is larger than the maximum array size specified by the module. SAS Micro Analytic Service rejects arrays larger than this size.	Supply an array with a size equal to or smaller than the maximum size allowed.
400	Cannot assign the value **specified_value** to the array element **parameter_name [element_number]** of type **type**.	An inappropriate value was received for the array parameter, for example, supplying a string value to an integer array parameter.	Specify appropriate matching values.
400	The value of field **field_name** must not be empty or missing.	The name and value parts of the parameter must be available to execute the module step.	Specify appropriate matching values.
400	The field **field_name** must not be repeated.	You cannot specify more than one value for a parameter.	Specify a single value.

HTTP Code	Error Message	Explanation	Remedy
400	Cannot parse the value of the data grid parameter **parameter_name**.	The value of a data grid parameter does not match the structure of a data grid.	Ensure that the supplied value of the parameter is appropriate for the data grid.
403	Cannot access the steps of the PRIVATE module **module_ID**.	The steps of private modules are not accessible.	Re-create the module as a public module.
403	Source code is only available for modules or submodules of type DS2.	Returned when trying to access the source code of a module that does not have source code.	Access the source code of a different module or submodule.
500	Cannot process the value of the data grid parameter **parameter_name** because it exceeds **integer_value** characters.	There is a size limit for a data grid. This is returned when the data grid is larger than the internal size of the data grid parameter.	Restructure the request data into multiple data grid objects.
500	Error **error_message** received when executing the step **step_ID** of the module **module_ID**.	An error was returned by the SAS Micro Analytic Service core during execution of the modules.	Search this appendix for information about the error message.
Transfer Errors			
400	The transfer object content is invalid. The content should not be modified after export.	The exported object appears to be corrupted or modified following the export.	Export the object from the source system again, and then retry the import.
400	The transfer object content is invalid for module **module_ID**. The content should not be modified after export.	The exported object for the given module appears to be corrupted or modified following the export.	Export the object from the source system again, and then retry the import.
Internal Errors			
500	Internal Error: The DS2 compiler encountered an unrecoverable error while compiling the module. Please check the log files for further diagnosis.	In most cases, this error is caused by insufficient stack space for the SAS Micro Analytic Service core TK subsystem.	In SAS Environment Manager, increase the value of the configuration parameter core.tktstacksizekbytes, as appropriate.
500	An internal error occurred. Please check the log files for further diagnosis.	This is a generic error that occurs during execution. It indicates that an unrecoverable situation has occurred.	Please contact SAS Technical Support.
500	An internal error occurred. Cannot load the file **file_name**. Please ensure that the file exists and the server is able to access it.	This is typically returned when the file containing the analytic store model is not accessible during execution of the module.	Please contact SAS Technical Support.

Appendix 4

Table Service Driver Reference

DB2 Driver Reference **167**
Understanding the Table Services Driver for DB2 167
Data Service Connection Options for DB2 168
DB2 Wire Protocol Driver Usage Notes 172

FedSQL Driver Reference **173**
Overview 173
Connection Options 173

ODBC Driver Reference **176**
About ODBC 176
Understanding the Table Services Driver for ODBC 176
Data Service Connection Options for ODBC 176
Wire Protocol Driver Usage Notes 182

Oracle Reference **183**
Understanding the Table Services Driver for Oracle 183
Data Service Connection Options for Oracle 183
Oracle Wire Protocol Driver Usage Notes 189

PostgreSQL Driver Reference **189**
Understanding the SAS Federation Server Driver for PostgreSQL 189
Data Service Connection Options for PostgreSQL 190

SAS Data Set Reference **194**
Overview 194
Understanding the Driver for Base SAS 194
Data Service Connection Options for SAS Data Sets 194

Teradata Reference **198**
Understanding the Table Services Driver for Teradata 198
Data Service Connection Options for Teradata 198

DB2 Driver Reference

Understanding the Table Services Driver for DB2

The table services driver for DB2 (driver for DB2) enables table services to read and update legacy DB2 tables. In addition, the driver creates DB2 tables that can be accessed by both table services and the DB2 database management system (DBMS).

The driver for DB2 supports most of the FedSQL functionality. The driver also enables an application to submit native DB2 SQL statements.

The table services driver for DB2 is a remote driver, which means that it connects to a server process in order to access data. The process might be running on the same machine as the table services driver, or it might be running on another machine in the network.

The table services driver for DB2 uses shared libraries that are referenced as shared objects in UNIX. You must add the location of the shared libraries to one of the system environment variables and, if necessary, specify the DB2 version that you have installed. Before setting the environment variables, as shown in the examples below, you must also set the following environment variables:

- The INSTHOME environment variable must be set to your DB2 home directory.
- The DB2DIR environment variable should also be set to the value of INSTHOME.
- The DB2INSTANCE environment variable should be set to the DB2 instance that was configured by the administrator.

```
AIX
Bourne Shell
$ LIBPATH=$INSTHOME/lib:$LIBPATH
$ export LIBPATH
C Shell
$ setenv LIBPATH $INSTHOME/lib:$LIBPATH
HP-UX and HP-UX for the Itanium Processor
  Family Architecture
Bourne Shell
$ SHLIB_PATH=$INSTHOME/lib:$SHLIB_PATH
$ export SHLIB_PATH
C Shell
$ setenv SHLIB_PATH $INSTHOME/lib:$SHLIB_PATH
Linux for Intel Architecture, Linux for x64, Solaris,
  and Solaris for x64
Bourne Shell
$LD_LIBRARY_PATH=$INSTHOME/lib:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
C Shell
$ setenv LD_LIBRARY_PATH $INSTHOME/lib:$LD_LIBRARY_PATH
```

Data Service Connection Options for DB2

Overview

The data service connection arguments for DB2 include connection options and advanced options.

Note: When performing connections through DSNs or connection strings, the FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

Connection Options

Connection options are used to establish a connection to a data source. Specify one or more connection options. Here is an example:

```
driver=sql;conopts=(driver=db2;uid=myuid;
```

```
pwd=Blue31;conopts=(DSN=MYDSN);CATALOG=TSSQL)
```

The table services driver for DB2 supports the following connection options for DB2 data sources.

Option	Description
CATALOG	**CATALOG=catalog-identifer;** Specifies an arbitrary identifier for an SQL catalog, which groups logically related schemas. Any identifier is valid (for example, **catalog=DB2**). You must specify a catalog. For the DB2 database, this is a logical catalog name to use as an SQL catalog identifier. *Note:* The FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.
DATABASE|DB	**DATABASE=database-specification;** Specifies the name of the DB2 database (for example, **database=sample, DB=sample**). *Note:* You must specify a database name.
DRIVER	**DRIVER=DB2;** Identifies the DB2 data source to which you want to connect. *Note:* You must specify the driver.

Advanced Connection Options

The table services driver for DB2 supports the following advanced connection options for DB2 data sources.

Option	Description
CLIENT_ENCODING	**CLIENT_ENCODING=encoding-value** Used to specify the encoding of the DB2CODEPAGE to the DB2 driver. When using this option, you must also set the DB2CODEPAGE environment variable on the client. When the encoding of the DB2 client layer (stored in DBCODEPAGE) is different from the encoding value of the DB2 operating system value, the DB2 client layer attempts to convert incoming data to the DB2 encoding value that is stored in DB2CODEPAGE. To prevent the client layer from converting data incorrectly, you must first determine the correct value for DB2CODEPAGE and then set the CLIENT_ENCODING= option to match the corresponding encoding value in DB2CODEPAGE. For example, suppose you are storing Japanese characters in a DB2 database, and the client machine where the DB2 driver is executing is a Windows machine that is running CP1252 encoding. When the application tries to extract the data into the table services driver, the DB2 client layer attempts to convert these Japanese characters into Latin1 representation, which does not contain Japanese characters. As a result, a garbage character appears in order to indicate a failure in transcoding. To resolve this situation, you must first set the DB2CODEPAGE environment variable value to 1208 (the IBM code page value that matches UTF-8 encoding). That enables you to specify that the DB2 client layer send the data to the application in UTF-8 instead of converting it into Latin1. In addition, you must specify the corresponding encoding value of DB2CODEPAGE because the table services driver for DB2 cannot derive this information from a DB2 session. For this particular Windows case, set the CLIENT_ENCODING= option to the UTF-8 encoding in order to match the DB2CODEPAGE value (1208) and also to specify the DB2CODEPAGE value to the DB2 driver. However, changing the value of DB2CODEPAGE affects all applications that run on that machine. You should reset the value to the usual DB2CODEPAGE value, which was derived when the database was created. *Note:* Setting the DB2CODEPAGE value or the CLIENT_ENCODING= value incorrectly can cause unpredictable results. You should set these values only when a situation such as the example above occurs. *Note:* You can specify any valid encoding value for CLIENT_ENCODING=option.
CT_PRESERVE	**CT_PRESERVE=STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.

Option	Description
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes that are supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**
DRIVER_TRACE	**DRIVER_TRACE='API | SQL | ALL';** Requests tracing information, which logs transaction records to an external file that can be used for debugging purposes. The driver writes a record of each command that is sent to the database to the trace log based on the specified tracing level, which determines the type of tracing information. Here are the tracing levels: • **API** Specifies that API method calls be sent to the trace log. This option is most useful if you are having a problem and need to send a trace log to SAS Technical Support for troubleshooting. • **SQL** Specifies that SQL statements that are sent to the database management system (DBMS) be sent to the trace log. Tracing information is DBMS specific, but most table services drivers log SQL statements such as SELECT and COMMIT. • **ALL** Activates all trace levels. • **DRIVER** Specifies that driver-specific information be sent to the trace log. **Default:** Tracing is not activated. *Note:* If you activate tracing, you must also specify the location of the trace log with DRIVER_TRACEFILE=. Note that DRIVER_TRACEFILE= is resolved against the TRACEFILEPATH set in ALTER SERVER. TRACEFILEPATH is relative to the server's content root location. (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=. **Interaction:** You can specify one trace level, or you can concatenate more than one by including the | (OR) symbol. For example, **driver_trace='api|sql'** generates tracing information for API calls and SQL statements.

Option	Description
DRIVER_TRACEFILE	**DRIVER_TRACEFILE='filename'**; Used to specify the name of the text file for the trace log. Include the file name and extension in single or double quotation marks (for example, **driver_tracefile='\mytrace.log'**). **Default:** The default TRACEFILE location applies to a relative file name, and it is placed relative to TRACEFILEPATH. **Requirement:** DRIVER_TRACEFILE is required when activating tracing using DRIVER_TRACE. **Interaction:** (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=.
DRIVER_TRACEOPTIONS	**DRIVER_TRACEOPTIONS=APPEND | THREADSTAMP | TIMESTAMP;** Specifies options in order to control formatting and other properties for the trace log: • **APPEND** Adds trace information to the end of an existing trace log. The contents of the file are not overwritten. • **TIMESTAMP** Prepends each line of the trace log with a time stamp. • **THREADSTAMP** Prepends each line of the trace log with a thread identification. **Default:** The trace log is overwritten with no thread identification or time stamp.
PASSWORD	**PWD=password** Specifies the password for DB2.
UID	**UID=user-id;** Specifies the DB2 login user ID.

DB2 Wire Protocol Driver Usage Notes

There are a number of third-party wire protocol ODBC drivers that communicate directly with a database server, without having to communicate through a client library. When you configure the ODBC drivers on Windows or UNIX, you can set certain options. SAS runs best when these options are selected. Some, but not all, are selected by default.

Windows	The options are located on the **Advanced** or **Performance** tabs in the ODBC Administrator.
UNIX	The options are available when configuring data sources using the ODBC Administrator tool. Values can also be set by editing the **odbc.ini** file in which their data sources are defined.

Note: A DSN configuration that uses a wire protocol driver with the catalog option selected returns only the schemas that have associated tables or views. To list all existing schemas, create a DSN without selecting the catalog option.

When configuring an ODBC DSN using the DB2 Wire Protocol driver, set the following advanced option:

- **Application Using Threads**

FedSQL Driver Reference

Overview

The FedSQL language driver supports the FedSQL dialect, as documented in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*. When loaded, the FedSQL driver parses SQL requests, and then sends the parsed query to the appropriate data source driver to determine whether the functionality can be handled by the data service. The FedSQL driver includes an SQL processor that supports the FedSQL dialect. The main emphasis of the FedSQL driver is to support federation of data sources. For example, if an SQL submission is requesting data from DB2 to be joined with data from Oracle, the SQL processor requests the data from the data sources and then performs the join. The FedSQL driver supports the FedSQL dialect regardless of the data source that it comes from. For example, if the SQL request is from a single data source that does not support a particular SQL function, the FedSQL processor guarantees implementation of the request.

Connection Options

- CONOPTS=((connection string 1);(connection string 2); ... (connection string <*n*>)) - Specifies one or more data source connection strings. For example, the following illustrates a federated connection string including Oracle, Teradata, Netezza, and Base SAS data sources:

```
driver=sql;conopts=((driver=oracle;catalog=acat;uid=myuid;
pwd=myPass9;path=oraclev11.abc.123.com:1521/ORA11G);
(driver=teradata;catalog=bcat;uid=model;
pwd='{sas002}C5DDFFF91B5D31DFFFCE9FFF';
server=terasoar;database=model);(driver=netezza;uid=myuid;
pwd=myPass2;server=mysrvr;database=testdb;catalog=(ccat={TEST}));
(driver=base;catalog=dcat;schema=(name=dblib;primarypath=/u/mypath/mydir)))
```

- DEFAULT_CATALOG=*catalog-name* - Used to specify the name of the catalog to set as the current catalog upon connecting. This option is useful for SQL Server connections and federated connections.
- DEFAULT_ATTR=(*attr=value*;...) - Used to specify connection handle or statement handle attributes supported for initial connect-time configuration., where **`attr=value`** corresponds to any of the following options:

SQL_CURSORS=*n*

FedSQL connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2.

- A value of 0 causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one.
- A value of 1 causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is never used.
- A value of 2 (default) causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available, otherwise the cursor is forward only.

```
DEFAULT_ATTR=(SQL_CURSORS=2)
```

SQL_AC_BEHAVIOR=*n*

FedSQL connection handle option. Specifies whether FedSQL should use transactions when processing complex operations (for example, **"CREATE TABLE xxx AS SELECT yyy FROM zzz"** or a multi-row delete statement that requires multiple operations to delete the underlying rows). Possible values are 0 (default), 1, and 2.

- A value of **0** (default) means that no transactions are attempted under-the-covers and operations such as emulated UPDATE, DELETE, or INSERT are not guaranteed to be atomic.
- A value of **1** means that FedSQL tries to use transactions to better support the correct behavior when AUTOCOMMIT is set to ON (where individual operations like UPDATE, DELETE, and INSERT should be atomic).
- A value of **2** means that transactions are required. This option fails if the underlying drivers do not support transactions.

```
DEFAULT_ATTR=(SQL_AC_BEHAVIOR=0)
```

SQL_MAX_COL_SIZE=*n*

FedSQL statement handle option. Enables a user to specify the size of the **varchar** or **varbinary** that is used for potentially truncated long data when direct bind is not possible.

- The default value is 32767.
- The limit for this size is 1 MG. If the value exceeds 1 MG, FedSQL resets the value and returns an **Option value changed** warning.

```
DEFAULT_ATTR=(SQL_MAX_COL_SIZE=1048576)
```

SQL_PUSHDOWN=*n*

FedSQL statement handle option. This option tells FedSQL if and when it should try to push down SQL to the underlying driver. The values are 8, 2, or 0 (default).

- A value of 8: (PLAN_FORCE_PUSHDOWN_SQL) - Complete statement pushdown is required. If that is not possible, the INSERT, UPDATE, DELETE, or CREATE TABLE AS statement fails.
- A value of 2: (PLAN_DISABLE_PUSHDOWN_SQL) - Specifies that the INSERT, UPDATE, DELETE, or CREATE TABLE AS statement not be pushed down to the underlying driver.
- A value of 0 (default): Specifies that the FedSQL processor determine whether the INSERT, UPDATE, DELETE, or CREATE TABLE AS statement should be pushed down to the underlying driver.

```
DEFAULT_ATTR=(SQL_PUSHDOWN=0)
```

SQL_STMT_MEM_LIMIT=*n*

FedSQL statement handle option. Used to control the amount of memory that is available to FedSQL to answer SQL requests.

- (*n*) is treated as an integer and is specified in bytes.
- The following example allows 200 MB of memory:

```
DEFAULT_ATTR=(SQL_STMT_MEM_LIMIT=209715200)
```

SQL_TXN_EXCEPTIONS=*n*

FedSQL connection handle option. Supports dynamic connections regardless of the specified transaction isolation. Possible values are 0 or 2 (default).

- Specify a value of 0 to disable support for dynamic connections.
- Specify a value of 2 to enable support for dynamic connections.

```
DEFAULT_ATTR=(SQL_TXN_EXCEPTIONS=2)
```

SQL_USE_EVP=*n*

FedSQL statement handle option. This option optimizes the driver for large result sets. The possible values are 0 or 1 (default) and are used as follows:

- Specify 0 to turn optimization OFF.
- Specify 1 to enable optimization (ON).

```
DEFAULT_ATTR=(SQL_USE_EVP=0)
```

SQL_VDC_DISABLE=*n*

FedSQL statement handle option. This option is used to allow or disallow use of cached data for a statement. The possible values are 0 (default) or 1 and are used as follows:

- Specify a value of 0 to enable cached data.
- Specify a value of 1 to disable cached data.

```
DEFAULT_ATTR=(SQL_VDC_DISABLE=1)
```

SQL_XCODE_WARN=*n*

FedSQL statement handle option. Used to warn when there is an error while transcoding data during row input or output operations. Possible values are 0 (default), 1, or 2 and are used as follows:

- Specify 0 to return an error if data cannot be transcoded.
- Specify 1 to return a warning if data cannot be transcoded.
- Specify 2 to ignore transcoding errors.

```
DEFAULT_ATTR=(SQL_XCODE_WARN=1)
```

ODBC Driver Reference

About ODBC

This section provides functionality details and guidelines for the open database connectivity (ODBC) databases that are supported by the table services driver for ODBC (driver for ODBC).

ODBC standards provide a common interface to a variety of databases, including dBASE, Microsoft Access, Oracle, Paradox, and Microsoft SQL Server databases. Specifically, ODBC standards define APIs that enable an application to access a database if both the application and the database conform to the specification. ODBC also provides a mechanism to enable dynamic selection of a database that an application is accessing. As a result, users can select databases other than those that are specified by the application developer.

Understanding the Table Services Driver for ODBC

The driver for ODBC enables table services to read and update legacy ODBC database tables. In addition, the driver creates tables that can be accessed by both table services and an ODBC database.

The driver for ODBC supports most of the FedSQL functionality. The driver also enables an application to submit native database-specific SQL statements.

The driver for ODBC is a remote driver, which means that it connects to a server process in order to access data. The process might be running on the same machine as table services, or it might be running on another machine in the network.

Data Service Connection Options for ODBC

Overview

To access data that is hosted on table services, a client must submit a connection string, which defines how to connect to the data. The data service connection arguments for an ODBC-compliant database include connection options and advanced connection options.

To configure ODBC data sources, you might have to edit the .odbc.ini file in your home directory. Some ODBC driver vendors allow system administrators to maintain a centralized copy, by setting the environment variable ODBCINI. For specific configuration information, see your vendor documentation. The table services driver for ODBC uses shared libraries that are referenced as shared objects in UNIX. You must add the location of the shared libraries to one of the system environment variables, so that drivers for ODBC are loaded dynamically at run time. You must also set the ODBCHOME environment variable to your ODBC home directory before setting the environment variables, as shown in the following example.

```
export ODBCHOME=/dbi/odbc/dd7.1.4
 export ODBCINI=/ODBC/odbc_714_MASTER.ini
 LD_LIBRARY_PATH=/dbi/odbc/dd7.1.4/lib:${LD_LIBRARY_PATH}
 export LD_LIBRARY_PATH=${LD_LIBRARY_PATH%:}
```

Connection Options

Connection options are used to establish a connection to a data source. Specify one or more connection options when defining a data service. Here is an example:

```
driver=sql;conopts=(driver=odbc;
catalog=acat;conopts=(dsn=ODBCPgresDD;pwd=Tester2))
```

The driver for ODBC supports the following connection options.

Option	Description
CATALOG	**CATALOG=catalog-identifier;** Specifies an arbitrary identifier for an SQL catalog, which groups logically related schemas. For databases that do not support native catalogs, any identifier is valid (for example, catalog=myodbc). For databases like Microsoft SQL Server that do support native catalogs, CATALOG= is not required. The connection defaults to CATALOG=* unless you specify a logical name for the catalog and map it to the native catalog name in the database. For example, to map the logical catalog **mycat** to the native catalog named **newusers**, use the following command: **catalog=(mycat=newusers);**. Catalog name maps can be used only with FedSQL. They are not valid with native SQL. *Note:* The FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

CONOPTS	**CONOPTS=(ODBC–compliant database connection string);**
	Specifies an ODBC-compliant database connection string using ODBC-style syntax. These options, combined with the ODBC_DSN option, must specify a complete connection string to the data source. If you include a DSN= or FILEDSN= specification within the CONOPTS= option, do not use the ODBC_DSN= connection option. However, you can specify the ODBC database-specific connection options by using CONOPTS=. Then you can specify an ODBC DSN that contains other connection information by using the ODBC_DSN= connection option.
	Here is an example string using the CONOPTS option:
	`driver=sql;conopts=((driver=odbc;catalog=acat;`
	`conopts=(dsn=ODBCPgresDD;pwd=Tester2));`
	`(driver=postgres;catalog=bcat;uid=myuid;pwd='123pass';`
	`server=sv.abc.123.com;port=5432;DB=mydb;schema=public))`

DRIVER	**DRIVER=ODBC**; Calls the table services driver for ODBC. This specifies that the data service to which you want to connect must be an ODBC-compliant database. *Note:* DRIVER is a required option. You must specify the driver.
ODBC_DSN	**ODBC_DSN=odbc dsn name** Specifies a valid ODBC-compliant database DSN that contains connection information for connecting to the ODBC-compliant database. You can use the CONOPTS= option in addition to ODBC_DSN= option to specify database-specific connection options not provided by table services. Do not specify the ODBC DSN in both CONOPTS= and ODBC_DSN=.

Advanced Connection Options

The driver for ODBC supports the following advanced connection options for an ODBC-compliant database.

Option	Description
CT_PRESERVE	**CT_PRESERVE = STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.
ENABLE_MARS	**ENABLE_MARS= NO|YES** Enables or disables the use of multiple active result sets (MARS) on Microsoft SQL Server. FedSQL cannot permit transactions on top of Microsoft SQL Server because Microsoft SQL Server allows only one cursor per transaction. Set this option to YES so that FedSQL can allow transactions under a given Microsoft SQL Server connection.

Option	Description
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**

Option	Description
DEFAULT_CURSOR_TYPE	**DEFAULT_CURSOR_TYPE=FORWARD_ONLY | KEYSET_DRIVEN | DYNAMIC | STATIC;** Specifies a valid default cursor type for new statements. These options are valid: **FORWARD_ONLY** Specifies a non-scrollable cursor that moves only forward through the result set. Forward-only cursors are dynamic in that all changes are detected as the current row is processed. If an application does not require scrolling, the forward-only cursor retrieves data quickly, with the least amount of overhead processing. **KEYSET_DRIVEN** Specifies a scrollable cursor that detects changes that are made to the values of rows in the result set but that does not always detect changes to deletion of rows and changes to the order of rows in the result set. A keyset-driven cursor is based on row keys, which are used to determine the order and set of rows that are included in the result set. As the cursor scrolls the result set, it uses the keys to retrieve the most recent values in the table. It is sometimes helpful to have a cursor that can detect changes in the rows of a result set. A keyset-driven cursor uses a row identifier rather than caching the entire row into memory. It therefore uses much less disk space than other row caching mechanisms. Deleted rows can be detected when a SELECT statement that references the bookmark, row ID, or key column values fails to return a row. **DYNAMIC** Specifies a scrollable cursor that detects changes that are made to the rows in the result set. All INSERT, UPDATE, and DELETE statements that are made by all users are visible through the cursor. The dynamic cursor is good for an application that must detect all concurrent updates that are made by other users. **STATIC** Specifies a scrollable cursor that displays the result set as it existed when the cursor was first opened. The static cursor provides forward and backward scrolling. If the application does not need to detect changes but requires scrolling, the static cursor is a good choice. *Note:* The application can still override this value, but if the application does not explicitly set a cursor type, this value will be in effect

Option	Description
DRIVER_TRACE	**DRIVER_TRACE='API | SQL | ALL';** Requests tracing information, which logs transaction records to an external file that can be used for debugging purposes. The driver writes a record of each command that is sent to the database to the trace log based on the specified tracing level, which determines the type of tracing information. Here are the tracing levels: • **ALL** Activates all trace levels. • **API** Specifies that API method calls be sent to the trace log. This option is most useful if you are having a problem and need to send a trace log to SAS Technical Support for troubleshooting. • **DRIVER** Specifies that driver-specific information be sent to the trace log. • **SQL** Specifies that SQL statements that are sent to the database management system (DBMS) be sent to the trace log. Tracing information is DBMS specific, but most table services drivers log SQL statements such as SELECT and COMMIT. **Default:** Tracing is not activated. *Note:* If you activate tracing, you must also specify the location of the trace log with DRIVER_TRACEFILE=. Note that DRIVER_TRACEFILE= is resolved against the TRACEFILEPATH set in ALTER SERVER. TRACEFILEPATH is relative to the server's content root location. (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=. **Interaction:** You can specify one trace level, or you can concatenate more than one by including the | (OR) symbol. For example, **driver_trace='api|sql'** generates tracing information for API calls and SQL statements.
DRIVER_TRACEFILE	**DRIVER_TRACEFILE='filename';** Used to specify the name of the text file for the trace log. Include the file name and extension in single or double quotation marks (for example, **driver_tracefile='\mytrace.log'**). **Default:** The default TRACEFILE location applies to a relative file name, and it is placed relative to TRACEFILEPATH. **Requirement:** DRIVER_TRACEFILE is required when activating tracing using DRIVER_TRACE. **Interaction:** (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=.
DRIVER_TRACEOPTIONS	**DRIVER_TRACEOPTIONS=APPEND | THREADSTAMP | TIMESTAMP;** Specifies options in order to control formatting and other properties for the trace log: • **APPEND** Adds trace information to the end of an existing trace log. The contents of the file are not overwritten. • **THREADSTAMP** Prepends each line of the trace log with a thread identification. • **TIMESTAMP** Prepends each line of the trace log with a time stamp. **Default:** The trace log is overwritten with no thread identification or time stamp.
USER	**USER=user-ID;** Specifies the user ID for logging on to the ODBC-compliant database, such as Microsoft SQL Server, with a user ID that differs from the default ID. *Note:* The alias is **UID=**.

Option	Description
PASSWORD	**PASSWORD=password;** Specifies the password that corresponds to the user ID in the database. *Note:* The alias is **PWD=**.

Here are example connection strings that use the table services driver for ODBC:

```
driver=sql;conopts=((driver=odbc;catalog=acat;
conopts=(dsn=ODBCPgresDD;pwd=Tester2));
(driver=postgres;catalog=bcat;uid=myuid;pwd='123pass';
server=sv.abc.123.com;port=5432;DB=mydb;schema=public))
```

This connection string specifies catalog name maps to access multiple catalogs on Microsoft SQL Server:

```
driver=odbc; uid=jfox; pw=mypw; odbc_dsn=mySQLdsn;
    catalog=(cat1=mycat; cat2=testcat; cat3=users;
```

Wire Protocol Driver Usage Notes

Overview

There are a number of wire protocol ODBC drivers that communicate directly with a database server, without having to communicate through a client library. When you configure the ODBC drivers on Windows or UNIX, you can set certain options. SAS runs best when these options are selected. Some, but not all, are selected by default.

Windows	The options are located on the **Advanced** or **Performance** tabs in the ODBC Administrator window.
UNIX	The options are available when configuring data sources using the ODBC Administrator tool. Values can also be set by editing the **odbc.ini** file in which their data sources are defined.

Note: A DSN configuration that uses a wire protocol driver with the catalog option selected returns only the schemas that have associated tables or views. To list all existing schemas, create a DSN without selecting the catalog option.

SQL Server and SQL Server Legacy

Configure the following **Advanced** options for the SQL Server Wire Protocol driver and the SQL Server Legacy Wire Protocol driver:

- **Application Using Threads**
- **Enable Quoted Identifiers**
- **Fetch TWFS as Time**
- **Fetch TSWTZ as Timestamp**

Note:

1. Significant performance improvements have been realized when using the SQL Server Legacy Wire Protocol driver, as compared to the SQL Server Wire Protocol driver.

2. The SQL Server Legacy Wire Protocol driver does not support transactions when it is used with FedSQL enabled because the driver allows only a single statement per connection while FedSQL requires multiple statements per connection when using transactions.

Oracle Reference

Understanding the Table Services Driver for Oracle

The table services driver for Oracle enables table services to read and update legacy Oracle tables. In addition, the driver creates Oracle tables that can be accessed by both table services and Oracle.

The driver for Oracle supports most of the FedSQL functionality. The driver also enables an application to submit native Oracle SQL statements.

The driver for Oracle is a remote driver, which means that it connects to a server process in order to access data. The process might be running on the same machine as the table services, or it might be running on another machine in the network.

The table services driver for Oracle uses shared libraries that are referenced as shared objects in UNIX. You must add the location of the shared libraries to one of the system environment variables, and set any other environment variables required by the Oracle client libraries. The following Bourne shell commands provide an example:

```
ORAENV_ASK=NO; export ORAENV_ASK
ORACLE_HOME=/dbi/oracle/11g; export ORACLE_HOME
SASORA=V9; export SASORA
PATH=$ORACLE_HOME/bin:/bin:/usr/bin:/usr/ccs/
   bin:/opt/bin:$PATH; export PATH
TMPDIR=/var/tmp; export TMPDIR
LD_LIBRARY_PATH=/usr/openwin/lib:$ORACLE_HOME/
   lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
TWO_TASK=oraclev11; export TWO_TASK
```

Data Service Connection Options for Oracle

Overview

To access data that is hosted on the table services, a client must submit a connection string, which defines how to connect to the data. The data service connection arguments for an Oracle server include connection options and advanced options.

Connection Options

Connection options are used to establish a connection to a data source. Specify one or more connection options. Here is an example:

```
driver=sql;conopts=(driver=oracle;
catalog=acat;uid=myuid;pwd=myPass9;
path=oraclev11.abc.123.com:1521/ORA11G)
```

The driver for Oracle supports the following connection options.

Option	Description
CATALOG	**CATALOG=catalog_identifier;** Specifies an arbitrary identifier for an SQL catalog, which groups logically related schemas. Any identifier is valid such as **catalog=oracle_test**. You must specify a catalog. For the Oracle database, this is a logical catalog name to use as an SQL catalog identifier. *Note:* The FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.
DRIVER	**DRIVER=ORACLE**; Identifies the data service to which you want to connect, which is an Oracle database. *Note:* You must specify the driver.
PATH	**PATH=database-specification;** Specifies the Oracle connect identifier. A connect identifier can be a net service name, a database service name, or a net service alias.
UID	**UID=user-id;** Specifies an optional Oracle user ID. If the user ID contains blanks or national characters, enclose it in quotation marks. If you omit an Oracle user ID and password, the default Oracle user ID **OPS$sysid** is used, if it is enabled.
PWD	**PWD=password;** Specifies an optional Oracle database password that is associated with the Oracle user ID. **PWD=** is always used with **UID=** and the associated password is case-sensitive. If you omit **PWD=**, the password for the default Oracle user ID **OPS$sysid** is used, if it is active.

Advanced Connection Options

The driver for Oracle supports the following advanced connection options.

Option	Description
CT_PRESERVE	**CT_PRESERVE = STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes that are supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**

Option	Description
DRIVER_TRACE	**DRIVER_TRACE='API | SQL | ALL'** Requests tracing information, which logs transaction records to an external file that can be used for debugging purposes. The driver writes a record of each command that is sent to the database to the trace log based on the specified tracing level, which determines the type of tracing information. Here are the tracing levels: • **ALL** Activates all trace levels. • **API** Specifies that API method calls be sent to the trace log. This option is most useful if you are having a problem and need to send a trace log to SAS Technical Support for troubleshooting. • **DRIVER** Specifies that driver-specific information be sent to the trace log. • **SQL** Specifies that SQL statements that are sent to the database management system (DBMS) be sent to the trace log. Tracing information is DBMS specific, but most table services drivers log SQL statements such as SELECT and COMMIT. **Default:** Tracing is not activated. *Note:* If you activate tracing, you must also specify the location of the trace log with DRIVER_TRACEFILE=. Note that DRIVER_TRACEFILE= is resolved against the TRACEFILEPATH set in ALTER SERVER. TRACEFILEPATH is relative to the server's content root location. (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=. **Interaction:** You can specify one trace level, or you can concatenate more than one by including the | (OR) symbol. For example, **driver_trace='api|sql'** generates tracing information for API calls and SQL statements.
DRIVER_TRACEFILE	**DRIVER_TRACEFILE='filename';** Used to specify the name of the text file for the trace log. Include the file name and extension in single or double quotation marks (for example, **driver_tracefile='\mytrace.log'**). **Default:** The default TRACEFILE location applies to a relative file name, and it is placed relative to TRACEFILEPATH. **Requirement:** DRIVER_TRACEFILE is required when activating tracing using DRIVER_TRACE. **Interaction:** (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=.
DRIVER_TRACEOPTIONS	**DRIVER_TRACEOPTIONS=APPEND | THREADSTAMP | TIMESTAMP;** Specifies options in order to control formatting and other properties for the trace log: • **APPEND** Adds trace information to the end of an existing trace log. The contents of the file are not overwritten. • **THREADSTAMP** Prepends each line of the trace log with a thread identification. • **TIMESTAMP** Prepends each line of the trace log with a time stamp. **Default:** The trace log is overwritten with no thread identification or time stamp.
ORA_ENCODING	**ORA_ENCODING=UNICODE;** Specifies that the Oracle data be returned in Unicode to table services. **UNICODE** is the default setting and is independent of the **NLS_LANG** environment variable setting.

Option	Description
ORNUMERIC	**ORANUMERIC=NO | YES** Specifies how numbers that are read from or inserted into the Oracle NUMBER column are treated. This option defaults to YES so that a NUMBER column with precision or scale is described as **TKTS_NUMERIC**. This option can be specified as both a connection option and a table option. When specified as both a connection and table option, the table option value overrides the connection option. • **NO** Indicates that the numbers are treated as **TKTS_DOUBLE** values. They might not have precision beyond 14 digits. • **YES** Indicates that non-integer values with explicit precision are treated as TKTS_NUMERIC values. This is the default setting.
USE_CACHED_CATALOG	**USE_CACHED_CATALOG=YES | NO;** Specifies whether to use the cached catalog rather than compiling a new catalog on every run. Setting this option to YES can improve the performance of the TKTSForeignKeys API. The default setting is YES. *Note:* Before you can use this option, you must complete the following steps: 1. Create a materialized view. See the example code in "Creating a Materialized View (USE_CACHED_CATALOG)" on page 188. 2. Use the ALTER DSN statement to add the USE_CACHED_CATALOG connection option.

Creating a Materialized View (USE_CACHED_CATALOG)

The following example shows you how to create a materialized view. Use this script if **USE_CACHED_CATALOG** is set to YES above.

```
/*----------------------SAS_CACHED_CATALOG.SQL-------------------------------*/
/* This script is used to create the materialized and the synonym needed to
   get the ForeignKey metadata.  Work with your DBA to set this up.
   Materialized views can be complex and so thorough understanding will help us
   use them effectively. Especially deciding how to do the refreshes.
   Here we provide the simplest possible steps to create the required materialized
   view and the command to refresh it manually. The materialized view below can
   be created in any schema with any name. Feel free to add whatever REFRESH
   options suits your purpose. Note that you might need additional steps based
   on the REFRESH option setting. Here we provide the simplest possible way to
   do this.  The PUBLIC synonym pointing to this Materialized view must be
   named "SAS_CACHED_FK_CATALOG_PSYN". This synonym must be visible to
   PUBLIC (or the set of users who will be needing Foreignkey metadata) so that
   it is accessible from any   schema.
*/

Create materialized view SAS_CACHED_FK_CATALOG_MATVIEW REFRESH ON DEMAND as SELECT
PKAC.OWNER as PKTABLE_SCHEM,
PKAC.TABLE_NAME as PKTABLE_NAME,
PKACC.COLUMN_NAME as PKCOLUMN_NAME,
FKAC.OWNER as FKTABLE_SCHEM,
FKAC.TABLE_NAME as FKTABLE_NAME,
FKACC.COLUMN_NAME as FKCOLUMN_NAME,
FKACC.POSITION as KEY_SEQ,
FKAC.CONSTRAINT_NAME as FK_NAME,
PKAC.CONSTRAINT_NAME as PK_NAME
from
sys.all_constraints PKAC, sys.all_constraints FKAC,
sys.all_cons_columns PKACC, sys.all_cons_columns FKACC

where

FKAC.r_constraint_name=PKAC.constraint_name and
FKAC.constraint_name=FKACC.constraint_name and
PKAC.constraint_name=PKACC.constraint_name and PKAC.constraint_type='P' and
FKAC.constraint_type='R' and FKAC.owner=FKACC.owner and PKAC.owner=PKACC.owner
and PKAC.table_name=PKACC.table_name and FKAC.table_name=FKACC.table_name and
FKACC.position = PKACC.position ;

/* The synonym name *must* be SAS_CACHED_FK_CATALOG_PUBLIC_SYNONYM */
create public synonym  SAS_CACHED_FK_CATALOG_PSYN for SAS_CACHED_FK_CATALOG_MATVIEW;
grant all on  SAS_CACHED_FK_CATALOG_PSYN to PUBLIC;

/*---------Manual REFRESH of the Materialized View----------------------------*/
/* Note there are several ways to do this, consult with your DBA.
   Here are a couple of ways:
*/
execute DBMS_MVIEW.REFRESH('SAS_CACHED_FK_CATALOG_MATVIEW');
execute DBMS_SNAPSHOT.REFRESH('SAS_CACHED_FK_CATALOG_MATVIEW', '?');
```

Oracle Wire Protocol Driver Usage Notes

Wire protocol ODBC drivers communicate directly with a database server without having to communicate through a client library. When you configure the ODBC drivers on Windows or UNIX, you can set certain options. SAS runs best when these options are selected. Some, but not all, are selected by default.

Windows	The options are located on the **Advanced** or **Performance** tabs in the ODBC Administrator.
UNIX	The options are available when you are configuring data sources using the ODBC Administrator tool. Values can also be set by editing the **`odbc.ini`** file in which their data sources are defined.

Note: When you use a wire protocol driver to create an ODBC connection, the following special considerations apply:

1. A DSN configuration that uses a wire protocol driver with the catalog option selected returns only the schemas that have associated tables or views. To list all existing schemas, create a DSN without selecting the catalog option.
2. Verify that the Enable Bulk Load option is active in the ODBC DSN for databases that support this option. The Enable Bulk Load option is not enabled by default in the newer wire protocol drivers. As a result, insert performance suffers.

When configuring an ODBC DSN using the Oracle Wire Protocol driver, set the following advanced options:

- **Application Using Threads**
- **Enable SQLDescribeParam**
- **Describe at Prepare**
- **Enable N-CHAR Support**
- **Enable Scrollable Cursors**

PostgreSQL Driver Reference

Understanding the SAS Federation Server Driver for PostgreSQL

The table services driver for PostgreSQL enables table services to read and update legacy PostgreSQL tables. In addition, the driver creates PostgreSQL tables that can be accessed by both the table services and the PostgreSQL data management system.

The driver for PostgreSQL supports most of the FedSQL functionality. The driver also enables an application to submit native SQL statements.

The driver for PostgreSQL is a remote driver, which means that it connects to a server process in order to access data. The process might be running on the same machine as the table services, or it might be running on another machine in the network.

The table services driver for PostgreSQL uses shared libraries that are referenced as shared objects in UNIX. You must add the location of the shared libraries to one of the system environment variables, and set any other environment variables required by the PostgreSQL client libraries. The following Korn shell commands provide an example:

```
LD_LIBRARY_PATH=/dbi/odbc/unixodbc2310/lib:/dbi/
   postgres/9.03.04/lib:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH%:}
export ODBCSYSINI=/dbi/postgres/9.03.04
export PATH=/dbi/postgres/9.03.04/bin:$PATH
unset LANG
export PGCLIENTENCODING=UTF8
```

Data Service Connection Options for PostgreSQL

Overview

To access data that is hosted on the table services, a client must submit a connection string, which defines how to connect to the data. The data service connection arguments for PostgreSQL include connection options and advanced options.

Connection Options

Connection options are used to establish a connection to a data source. Specify one or more connection options when defining a data service. Here is an example:

```
driver=sql;conopts=(driver=postgres;catalog=acat;
uid=myuid;pwd='123pass';server=sv.abc.123.com;
port=5432;DB=mydb;schema=public)
```

The following connection options are supported for PostgreSQL data sources.

Option	Description
CATALOG	**CATALOG=catalog-identifier;** Specifies an arbitrary identifier for an SQL catalog, which groups schemas that are logically related (for example, **catalog=ptgtest**). *Note:* The FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

CONOPTS	**CONOPTS=(ODBC–compliant database connection string);**
	Specifies an ODBC-compliant database connection string using ODBC-style syntax. These options, combined with the ODBC_DSN option, must specify a complete connection string to the data source. If you include a DSN= or FILEDSN= specification within the CONOPTS= option, do not use the ODBC_DSN= connection option. However, you can specify the ODBC database-specific connection options by using CONOPTS=. Then you can specify an ODBC DSN that contains other connection information by using the ODBC_DSN= connection option.
	Here is an example string using the CONOPTS option:
	`driver=sql;conopts=`
	`((driver=odbc;catalog=acat;conopts=(dsn=ODBCPgresDD;pwd=Tester2));`
	`(driver=postgres;catalog=bcat;uid=myuid2;pwd='123mypass';`
	`server=sv.abc.123.com;port=5432;DB=mydb;schema=public))"`

Option	Description
DRIVER	**DRIVER=postgres**; Specifies the data service for the PostgreSQL database to which you want to connect. *Note:* DRIVER is a required option. You must specify a driver.
DATABASE	**DATABASE=database-name;** Specifies the name of the PostgreSQL database. Enclose the database name in single quotation marks if it contains spaces or non-alphanumeric characters. You can also specify DATABASE= with the DB= alias.**database=sample, DB=sample**.
DSN	**DSN=data-source-identifier;** Specifies the data source name to which you want to connect.
PWD	**PWD=password;** Specifies the password associated with the user ID. Enclose password in single quotation marks if it contains spaces or non-alphanumeric characters. You can also specify PASSWORD= with the PWD=, PASS=, and PW= aliases.
PORT	**PORT=port_number** Specifies the port number that is used to connect to the specified PostgreSQL Server. If you do not specify a port, the default is 5432.
SERVER	**SERVER='server-name'** Specifies the server name or IP address of the PostgreSQL server to which you want to connect. Enclose the server name in single quotation marks if the name contains spaces or non-alphanumeric characters: **SERVER='server name'**.
USER	**USER=user-name** Specifies the PostgreSQL user name (also called the user ID) that you use to connect to your database. If the user name contains spaces or non-alphanumeric characters, you must enclose it in quotation marks.

Advanced Options

The following advanced options are supported for PostgreSQL data sources.

Option	Description
ALLOW_UNQUOTED_NAMES	**ALLOW_UNQUOTED_NAMES=NO | YES** Specifies whether to enclose table and column names in quotation marks. Tables and columns are quoted when this option is set at NO. If set to YES, the driver does not automatically add quotation marks to table and column names if they are not specified. This allows PostgreSQL tables and columns to be created in the default lowercase. The default option is NO.
CLIENT_ENCODING	**CLIENT_ENCODING=cei** Used to specify encoding for the client.

Option	Description
CT_PRESERVE	**CT_PRESERVE=STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**

Option	Description
DRIVER_TRACE	**DRIVER_TRACE='API | SQL | ALL';** Requests tracing information, which logs transaction records to an external file that can be used for debugging purposes. The driver writes a record of each command that is sent to the database to the trace log based on the specified tracing level, which determines the type of tracing information. Here are the tracing levels: • **ALL** Activates all trace levels. • **API** Specifies that API method calls be sent to the trace log. This option is most useful if you are having a problem and need to send a trace log to SAS Technical Support for troubleshooting. • **DRIVER** Specifies that driver-specific information be sent to the trace log. • **SQL** Specifies that SQL statements that are sent to the database management system (DBMS) be sent to the trace log. Tracing information is DBMS specific, but most table services drivers log SQL statements such as SELECT and COMMIT. **Default:** Tracing is not activated. *Note:* If you activate tracing, you must also specify the location of the trace log with DRIVER_TRACEFILE=. Note that DRIVER_TRACEFILE= is resolved against the TRACEFILEPATH set in ALTER SERVER. TRACEFILEPATH is relative to the server's content root location. (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=. **Interaction:** You can specify one trace level, or you can concatenate more than one by including the | (OR) symbol. For example, **driver_trace='api|sql'** generates tracing information for API calls and SQL statements.
DRIVER_TRACEFILE	**DRIVER_TRACEFILE='filename'**; Used to specify the name of the text file for the trace log. Include the file name and extension in single or double quotation marks (for example, **driver_tracefile='\mytrace.log'**). **Default:** The default TRACEFILE location applies to a relative file name, and it is placed relative to TRACEFILEPATH. **Requirement:** DRIVER_TRACEFILE is required when activating tracing using DRIVER_TRACE. **Interaction:** (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=.
DRIVER_TRACEOPTIONS	**DRIVER_TRACEOPTIONS=APPEND | THREADSTAMP | TIMESTAMP;** Specifies options in order to control formatting and other properties for the trace log: • **APPEND** Adds trace information to the end of an existing trace log. The contents of the file are not overwritten. • **THREADSTAMP** Prepends each line of the trace log with a thread identification. • **TIMESTAMP** Prepends each line of the trace log with a time stamp. **Default:** The trace log is overwritten with no thread identification or time stamp.
MAX_BINARY_LEN	**MAX_BINARY_LEN=value;** Specifies a value, in bytes, that limits the length of long binary fields (LONG VARBINARY). Unlike other databases, PostgreSQL does not have a size limit for long binary fields. The default is 1048576.

Option	Description
MAX_CHAR_LEN	**MAX_CHAR_LEN=value;** Specifies a value that limits the length of character fields (CHAR and VARCHAR). The default is 2000.
MAX_TEXT_LEN	**MAX_TEXT_LEN=value;** Specifies a value that limits the length of long character fields (LONG VARCHAR). The default is 409500.
SCHEMA	**SCHEMA=value;** Specifies the default schema for the connection. If not specified, the schema, or list of schemas, is determined based on the value of the schema search path that is defined on the database server.
STRIP_BLANKS	**STRIP_BLANKS=YES|NO;** Specifies whether to strip blanks from character fields.

SAS Data Set Reference

Overview

The SAS data set is a SASProprietary file format, which contains data values that are organized as a table of rows (SAS observations) and columns (SAS variables). A supported SAS data set uses the extension **.sas7bdat**.

Understanding the Driver for Base SAS

The table services driver for Base SAS is a SASProprietary driver that provides Read and Update access to legacy SAS data sets. With the table services driver for Base, you can create SAS data sets that can be accessed by both the legacy and the table services data access services.

The driver supports much of the Base SAS functionality, such as SAS indexing and general integrity constraints, as well as much of the Federated Query Language (FedSQL) functionality.

The table services driver for Base SAS is an in-process driver, which means that it accesses data in the same process that executes the data access services. All server connections that are made with the table services driver for Base SAS use LOCKTABLE=SHARED and PATH_BIND=ACCESS connection options.

Data Service Connection Options for SAS Data Sets

Connection Options

To access data that is hosted on the table services, a client must submit a connection string, which defines how to connect to the data. The data service connection arguments for a SAS data set include connection options and advanced options. Here is an example:

```
driver=sql;conopts=(driver=base;catalog=acat;
schema=(name=dblib;primarypath=/u/path/mydir))
```

The following connection options are supported for SAS data sets:

Option	Description
CATALOG	**CATALOG=catalog-identifier;** Specifies an arbitrary identifier for an SQL catalog, which groups logically related schemas. A catalog name can be up to 32 characters long. You must specify a catalog. *Note:* The FedSQL language processor automatically quotes SQL identifiers that do not meet the regular naming convention as defined in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.
DRIVER	**DRIVER=BASE;** Identifies the data service to which you want to connect, which is a SAS data set. *Note:* You must specify DRIVER=BASE to access a SAS data set.
(SCHEMA) NAME	**NAME=schema-identifier;** Specifies an arbitrary identifier for an SQL schema. Any identifier is valid (for example, name=myfiles). The schema identifier is an alias for the physical location of the SAS library, which is much like the Base SAS libref. A schema name must be a valid SAS name and can be up to 32 characters long. You must specify a schema identifier.
PRIMARY PATH	**PRIMARYPATH=physical-location;** Specifies the physical location for the SAS library, which is a collection of one or more SAS files. For example, in directory-based operating environments, a SAS library is a group of SAS files that are stored in the same directory. *Note:* You must specify a primary path.
SCHEMA (ATTRIBUTES)	**SCHEMA=(attributes);** Specifies schema attributes that are specific to a SAS data set. A schema is a data container object that groups tables. The schema contains a name, which is unique within the catalog that qualifies table names. For a SAS data set, a schema is similar to a SAS library, which is a collection of tables with assigned attributes.

Advanced Options

Advanced driver options are additional options that are not required in order to connect to the data source. They are used to establish connections to catalogs, data source names (DSNs), and schemas. Although advanced options can also be used when connecting to a data service, doing so causes the specified options to apply to all data service connections.

The following advanced options are supported for SAS data sets:

Option	Description
ACCESS	**ACCESS=READONLY | TEMP;** • **READONLY** Assigns a read-only attribute to the schema. You cannot open a SAS data set to update or write new information. • **TEMP** specifies that the SAS data sets be treated as scratch files. That is, the system will not consume CPU cycles to ensure that the files do not become corrupted. ***TIP*** Use ACCESS=TEMP to save resources only when the data is recoverable. If TEMP is specified, data in memory might not be written to disk on a regular basis. This saves I/O, but could cause a loss of data if there is a crash.
CT_PRESERVE	**CT_PRESERVE = STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.
COMPRESS	**COMPRESS=NO | YES | CHAR | BINARY;** Controls the compression of rows in created SAS data sets. • **NO** Specifies that the rows in a newly created SAS data set are uncompressed (fixed-length records). This setting is the default. • **YES | CHAR** Specifies that the rows in a newly created SAS data set are compressed (variable-length records) by using RLE (Run Length Encoding). RLE compresses rows by reducing repeated consecutive characters (including blanks) to two- or three-byte representations. ***TIP*** Use this compression algorithm for character data. • **BINARY** Specifies that the rows in a newly created SAS data set are compressed (variable-length records) by using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file. ***TIP*** This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric columns). Because the compression function operates on a single record at a time, the record length must be several hundred bytes or larger for effective compression.

Option	Description
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes that are supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**
ENCODING	**ENCODING=encoding-value;** Overrides and transcodes the encoding for input or output processing of SAS data sets. *Note:* The default value is the current operating system setting.
LOCKTABLE	**LOCKTABLE=SHARED | EXCLUSIVE** Places exclusive or shared locks on SAS data sets. You can lock tables only if you are the owner or have been granted the necessary privilege. The default value for the table services is SHARED. • **SHARED** Locks tables in shared mode, allowing other users or processes to read data from the tables, but preventing other users from updating. • **EXCLUSIVE** Locks tables exclusively, preventing other users from accessing any table that you open.
PATH_BIND	**PATH_BIND=CONNECT | ACCESS** Specifies when and how schemas are validated during connection. CONNECT validates the entire connection string at the time of connection and returns an error if one or more schemas is invalid. ACCESS validates schemas when they are accessed so that processing continues regardless of errors in the schema portion of the connection string. ACCESS is the default for the table services.

Teradata Reference

Understanding the Table Services Driver for Teradata

The table services driver for Teradata provides Read and Update access to Teradata database tables and creates tables that can be accessed by both table services and Teradata.

The table services driver for Teradata supports most of the FedSQL functionality. The driver also enables an application to submit native Teradata SQL statements.

The table services driver for Teradata is a remote driver, which means that it connects to a server process to access data. The process might be running on the same machine as the table services, or it might be running on another machine in the network.

The table services driver for uses shared libraries that are referenced as shared objects in UNIX. You must add the location of the shared libraries to one of the system environment variables, and set any other environment variables that are required by the Teradata client libraries. The following Korn shell commands provide an example:

```
LD_LIBRARY_PATH=/opt/teradata/client/14.10/
   lib64:/opt/teradata/client/14.10/tbuild/lib64:/
   opt/teradata/client/14.10/tdicu/lib64:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH%:}
export COPERR=/opt/teradata/client/14.10/lib
export COPLIB=/opt/teradata/client/14.10/lib
export NLSPATH=/opt/teradata/client/14.10/tbuild/msg64/%N
```

Data Service Connection Options for Teradata

Connection Options

Connection options are used to establish a connection to a data source. Specify one or more connection options when defining a data service. Here is an example:

```
driver=sql;conopts=(driver=teradata;catalog=acat;
uid=myuid;pwd='{sas002}C5DDFFF91B5D31DFFFCE9FFF';
server=terasoar;database=model)
```

The following connection options are supported for a Teradata database.

Option	Description
CATALOG	**CATALOG=catalog-identifier;** Specifies an arbitrary identifier for an SQL catalog, which groups logically related schemas. Any identifier is valid (for example, catalog=tera). *Note:* You must specify a catalog.
DATABASE	**DATABASE=database-name;** Specifies the Teradata database. If you do not specify DATABASE=, you connect to the default Teradata database, which is often named the same as your user ID. If the database value that you specify contains spaces or non-alphanumeric characters, you must enclose it in quotation marks.

Option	Description
DRIVER	**DRIVER=TERA;** Identifies the data service to which you want to connect, which is a Teradata database. *Note:* You must specify the driver.
SERVER	**SERVER=server-name;** Specifies the Teradata server identifier.

Advanced Connection Options

The following advanced options are supported for Teradata database.

Option	Description
ACCOUNT	**ACCOUNT=account-ID;** Specifies an optional account number that you want to charge for the Teradata session.
CLIENT_ENCODING	**CLIENT_ENCODING=encoding-value** Used to specify the character set for the session. UTF8 is the default if encoding is not specified. These character sets are supported: ASCII EBCDIC EBCDIC037_0E KATAKANAEBCDIC KANJIEUC_0U LATIN9_0A THAI874_4A0 LATIN1250_1A0 CYRILLIC1251_2A0 LATIN1254_7A0 HEBREW1255_5A0 ARABIC1256_6A0 LATIN1258_8A0 TCHBIG5_1R0 SCHINESE936_6R0 KANJI932_1S0 HANGUL949_7R0 TCHINESE950_8R0 LATIN1252_3A0 SCHEBCDIC935_2IJ TCHEBCDIC937_3IB HANGULEBCDIC933_1II EBCDIC273_0E EBCDIC277_0E KANJIEBCDIC5035_0I KANJIEBCDIC5026_0I UTF8 UTF16

Option	Description
CT_PRESERVE	**CT_PRESERVE = STRICT | SAFE | FORCE | FORCE_COL_SIZE** Enables users to control how data types are mapped. Note that data type mapping is disabled when CT_PRESERVE is set to STRICT. If the requested type does not exist on the target database, an error is returned. Here are the options: • **STRICT** The requested type must exist in the target database. No type promotion occurs. If the type does not exist, an error is returned. • **SAFE** Target data types are upscaled only if they do not result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure all characters can be stored in the new encoding. • **FORCE** This is the default for all drivers. The best corresponding target data type is chosen, even if it could potentially result in a loss of precision or scale. When character encodings are changed, the new column size is recalculated to ensure that all characters can be stored in the new encoding. • **FORCE_COL_SIZE** This option is the same as FORCE, except that the column size for the new encoding is the same as the original encoding. This option can be used to avoid column size creep. However, the resulting column might be too large or too small for the target data.
DEFAULT_ATTR	**DEFAULT_ATTR=(attr=value;...)** Used to specify connection handle or statement handle attributes supported for initial connect-time configuration, where **attr=value** corresponds to any of the following options: • **CURSORS=n**- Connection handle option. This option controls the driver's use of client-side, result set cursors. The possible values are 0, 1, or 2. 0 Causes the driver to use client-side static cursor emulation if a scrollable cursor is requested but the database server cannot provide one. 1 Causes the driver to always use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is not used. 2 (Default) Causes the driver to never use client-side static cursor emulation if a scrollable cursor is requested. The database server's native cursor is used if available. Otherwise, the cursor is forward-only. Example: **DEFAULT_ATTR=(CURSORS=2)** • **USE_EVP=n** - Statement handle option. This option optimizes the driver for large result sets. The possible values are 0 (OFF) or 1 (ON), which is the default. Example: **DEFAULT_ATTR=(USE_EVP=0)** • **XCODE_WARN=n** - Statement handle option. Used to warn about possible character transcoding errors that occur during row input or output operations. Possible values are 0 (returns an error), 1 (returns a warning), or 2 (ignore transaction errors). 0 is the default. Example: **DEFAULT_ATTR=(XCODE_WARN=1)**

Option	Description
DRIVER_TRACE	**DRIVER_TRACE='API | SQL | ALL';** Requests tracing information, which logs transaction records to an external file that can be used for debugging purposes. The driver writes a record of each command that is sent to the trace log based on the specified tracing level, which determines the type of tracing information. Here are the tracing levels: • **ALL** Activates all trace levels. • **API** Specifies that API method calls be sent to the trace log. This option is most useful if you are having a problem and need to send a trace log to SAS Technical Support for troubleshooting. • **DRIVER** Specifies that driver-specific information be sent to the trace log. • **SQL** Specifies that SQL statements that are sent to the database management system (DBMS) be sent to the trace log. Tracing information is DBMS specific, but most table services drivers log SQL statements such as SELECT and COMMIT. **Default:** Tracing is not activated. *Note:* If you activate tracing, you must also specify the location of the trace log with DRIVER_TRACEFILE=. Note that DRIVER_TRACEFILE= is resolved against the TRACEFILEPATH set in ALTER SERVER. TRACEFILEPATH is relative to the server's content root location. (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=. **Interaction:** You can specify one trace level, or you can concatenate more than one by including the | (OR) symbol. For example, **driver_trace='api|sql'** generates tracing information for API calls and SQL statements.
DRIVER_TRACEFILE	**DRIVER_TRACEFILE='filename';** Used to specify the name of the text file for the trace log. Include the file name and extension in single or double quotation marks (for example, **driver_tracefile='\mytrace.log'**). **Default:** The default TRACEFILE location applies to a relative file name, and it is placed relative to TRACEFILEPATH. **Requirement:** DRIVER_TRACEFILE is required when activating tracing using DRIVER_TRACE. **Interaction:** (Optional) You can control trace log formatting with DRIVER_TRACEOPTIONS=.
DRIVER_TRACEOPTIONS	**DRIVER_TRACEOPTIONS=APPEND | THREADSTAMP | TIMESTAMP;** Specifies options in order to control formatting and other properties for the trace log: • **APPEND** Adds trace information to the end of an existing trace log. The contents of the file are not overwritten. • **THREADSTAMP** Prepends each line of the trace log with a thread identification. • **TIMESTAMP** Prepends each line of the trace log with a time stamp. **Default:** The trace log is overwritten with no thread identification or time stamp.
PASSWORD	**PASSWORD=password;** Specifies a Teradata password. The password must match your USER= value. The alias is PWD=. *Note:* You must specify the PASSWORD= option.

Option	Description
ROLE	**ROLE=security-role;** Specifies a security role for the session.
USER	**USER=user-id;** Specifies a Teradata user ID. If the ID contains blanks or national characters, enclose it in quotation marks. The alias is UID=. *Note:* You must specify the USER= option.

Appendix 5

SAS Micro Analytic Service Tuning Guidelines

Proper tuning can significantly improve the performance of SAS Micro Analytic Service. However, there is no single set of configuration recommendations that is ideal for all the models that SAS Micro Analytic Service can execute. This appendix explains the principles that you can apply to tune for optimal performance.

SAS Micro Analytic Service Configuration

Eventing and Authorization

To achieve high performance with SAS Micro Analytic Service, you should disable eventing and authorization. When these options are enabled, the throughput performance (transactions per second or *TPS*) is dramatically reduced. Disabling these options is the most beneficial configuration change that you can make to improve performance.

To disable eventing and authorization, apply the values shown to the following settings in SAS Environment Manager:

- **java_option_auth**: `-Dsas.authorization`=*false*
- **java_option_auth_remote**: `–Dsas.authorization.remote`=*false*
- **java_option_eventing**: `-Dsas.event.enabled`=*false*

Core Threads

In general, adding core threads increases throughput for a given load. However, for each core thread that is added, the marginal gains in throughput diminish sharply.

For optimal performance, it is recommended that you set the number of core threads equal to the number of cores on the server.

SAS Micro Analytic Service instances with a greater number of core threads tend to be more resilient to sub-optimal Apache Tomcat tuning conditions. Because of this, tuning is particularly important for SAS Micro Analytic Service instances with fewer than 40 threads.

Apache Tomcat Tuning

Most of the default Apache Tomcat settings are sufficient to achieve high throughput with SAS Micro Analytic Service. However, there is one setting that you can adjust to help increase performance—the maxThreads setting.

The maxThreads value is influenced by both the number of SAS Micro Analytic Service core threads and the nature of the workload.

SAS Micro Analytic Service instances with a low number of core threads (fewer than 20) typically perform best with a low maxThreads value. In this case, an appropriate value could be between 20 and 100.

Alternatively, the maxThreads value is not as important when executing complex models with many SAS Micro Analytic Service core threads (more than 20).

Note: This is the case for the BigDecision2 model shown in the example table below.

Here are some examples of optimal maxThread tunings:

Model	Model Complexity	Core Threads	Optimal MaxThreads Value
BigDecision1	complex	5	30
BigDecision2	complex	80	100
SimpleEcho1	trivial	5	50
SimpleEcho2	trivial	80	75

Apache HTTP Server Configuration

Multi-Processing Modules

The Apache HTTP server multi-processing modules (MPM) bind network ports on the system, accept requests, and dispatch child processes to handle the requests.

By default, MPMs are set to prefork mode, which corresponds to one process per connection. This is not ideal for SAS Micro Analytic Service performance.

Therefore, it is recommended that you set MPMs to worker mode, which enables each child process to have multiple threads. This setting has demonstrated better average throughput with less variance.

ServerLimit and MaxClients

The ServerLimit and MaxClients settings determine the number of Apache processes and the number of threads for each process, respectively.

Increasing either of these parameters causes Apache to use more memory. This is not a problem unless Apache exceeds the available memory and writes to disk, which is detrimental to performance.

It is recommended that you scale both settings to the largest values that will not exceed the available RAM at load time.

Appendix 6

Applying a New License

You must apply a new SAS license when your current SAS Micro Analytic Service license expires.

1. On the machine where the SAS Micro Analytic Service is deployed, log on with a user account that meets the requirements to deploy software. Those requirements are specified in "User and Group Requirements" in *SAS Viya for Linux: Deployment Guide*.
2. Move the current license files into a backup location. Your current license file (named setinit.txt) resides in the following location:

   `/opt/sas/viya/config/etc/SASMicroAnalyticService`
3. SAS distributes renewal licenses to customers as file attachments in a renewal order email (ROE). Make sure that your new license files (a .txt file and a .jwt file) reside in location that is accessible from your SAS Micro Analytic Service machine.

   Note: Some SAS Viya products use the text file (.txt). Other products use the JSON web token file (.jwt). SAS Micro Analytic Service uses the .txt file.
4. Copy the new license file to the following location:

   `/opt/sas/viya/config/etc/SASMicroAnalyticService`
5. Restart SAS Micro Analytic Service to apply the new license file.

Recommended Reading

- *SAS Intelligent Decisioning: User's Guide*
- *SAS Intelligent Decisioning: Administrator's Guide*
- *SAS Event Stream Processing: Overview*
- *SAS Event Stream Processing: Using Streaming Analytics*
- *SAS Event Stream Processing: Using SAS Event Stream Processing with Other Applications*
- *SAS Event Stream Processing: Visualizing Event Streams with Streamviewer*
- *SAS Event Stream Processing: Connectors and Adapters*
- *SAS Event Stream Processing: Publish/Subscribe API*
- *SAS Event Stream Processing: Programming Reference*
- *SAS Event Stream Processing: XML Language Dictionary*
- *SAS Event Stream Processing: Creating and Using Windows*
- *SAS Event Stream Processing: Using the ESP Server*
- *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*
- *SAS Event Stream Processing: Tutorials and Examples*
- *SAS DS2 Programmer's Guide*
- *SAS DS2 Language Reference*
- *SAS Viya Administration: Tuning*
- *SAS Viya Administration: Logging*
- *Encryption in SAS Viya: Data in Motion*
- *Encryption in SAS Viya: Data at Rest*
- *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*

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

A

analytic store models
  and Event Stream Processing 35
  and Intelligent Decisioning 89
architecture, SAS Micro Analytic Service 8
array, data type conversions 32
asynchronous execution 124

C

component hierarchy 7
composite modules 37
content, moving 134
contexts 6

D

database access
  DS2 122
deploying SAS Micro Analytic Service
  SAS Event Stream Processing 70
drivers
  Base SAS 194
  DB2 167
  FedSQL 173
  ODBC 176, 182
  Oracle 183
  PostgreSQL 189
  Teradata 198
  wire protocol 182
DS2
  calls between modules 33, 84
  character restrictions 28, 78
  connection strings 122
  database access 122
  database drivers 123
  executing in SAS Micro Analytic Service 33, 84
  HTTP package 134
  I/O 122
  managing large modules 86
  module compilation clustered environment 88
  module compilation minimum memory 88
  module compilation time-out value 87
  module loading time-out value 87
  stateful data 39, 93
  user-defined formats 28, 79
DS2 argument types
  SAS Event Stream Processing 31
  SAS Intelligent Decisioning 82
DS2 best practices
  SAS Event Stream Processing 51
  SAS Intelligent Decisioning 105
DS2 character-to-numeric conversions
  SAS Event Stream Processing 55
  SAS Intelligent Decisioning 109
DS2 global packages
  SAS Event Stream Processing 52
  SAS Intelligent Decisioning 106
DS2 hash package
  SAS Event Stream Processing 55
  SAS Intelligent Decisioning 109
DS2 invariant computations
  SAS Event Stream Processing 56
  SAS Intelligent Decisioning 110
DS2 local packages
  SAS Event Stream Processing 52
  SAS Intelligent Decisioning 106
DS2 methods
  SAS Intelligent Decisioning 79
DS2 methods and packages
  SAS Event Stream Processing 28
DS2 passing character values to methods
  SAS Event Stream Processing 56
  SAS Intelligent Decisioning 109
DS2 programming
  SAS Event Stream Processing 26
  SAS Intelligent Decisioning 76
DS2 return results
  SAS Event Stream Processing 51
  SAS Intelligent Decisioning 105
DS2 single computation
  SAS Event Stream Processing 56
  SAS Intelligent Decisioning 110

G

global packages, DS2
  SAS Event Stream Processing 52
  SAS Intelligent Decisioning 106

L

license, updating 205
local packages, DS2
  SAS Event Stream Processing 52
  SAS Intelligent Decisioning 106
lockdown mode 117
logging SAS Micro Analytic Service
  SAS Event Stream Processing 69
  SAS Intelligent Decisioning 131

M

MASCall package 33, 84
MASState package 39, 93
  categories of 40, 94
  number of methods in 42, 96
MASTktstacksize environment variable 70
microanalyticservice.conf file 123
modules
  context 6
  create and update time-out 86
  Python 112
  understanding 5

N

NULL key fields, derived event suppression rules 21, 23

P

packages
  MASCall 33, 84
  MASState 39, 93
  PyMAS 140
publishing DS2 source code
  SAS Event Stream Processing 26
  SAS Intelligent Decisioning 76
Python
  and SAS lockdown mode 117
  compiling modules 117
  configuring for Intelligent Decisioning 118
  configuring for SAS Event Stream Processing 66
  modules 112
  return values 63, 114
Python, public and private methods
  SAS Event Stream Processing 62
  SAS Intelligent Decisioning 113

R

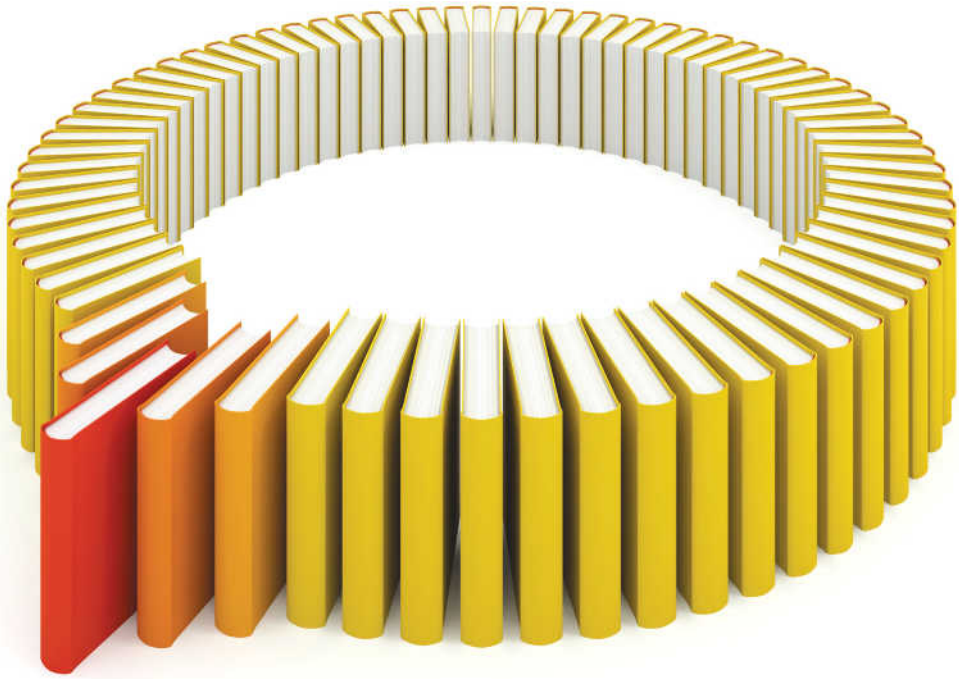
REST server error messages 161
restricted characters, DS2 28, 78
return codes 149
revisions 7

S

SAS Environment Manager 125
SAS Event Stream Processing environment
  analytic store models 35
  data type mappings 15
  DS2 argument types 31
  DS2 methods and packages 28
  elements 12
  event operation codes and flags 19
  generating multiple derived events 21
  overview 3
SAS Intelligent Decisioning environment
  analytic store models 89
  DS2 argument types 82
  DS2 methods 79
  overview 3
SAS Micro Analytic Score service 3
SAS Micro Analytic Service
  architecture 8
  authorization 133
  component hierarchy 7
  concepts 5
  configuration 125
  contexts 6
  logging, in SAS Event Stream Processing 69
  logging, in SAS Intelligent Decisioning 131
  modules 6
  REST server error messages 161
  return codes 149
  revision, for a module 7
  starting and stopping 124
SAS Model Manger 3
scalar, data type conversions 32
SCAN, replacing in DS2 code
  SAS Event Stream Processing 53
  SAS Intelligent Decisioning 107
stateful data
  methods 40, 94
  operations 39, 93
synchronous execution 124

T
timed execution 124
transfer service 134
TRANWRD, replacing in DS2 code
  SAS Event Stream Processing 53
  SAS Intelligent Decisioning 107
  tuning 203

U
user-defined formats
  DS2 28, 79



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.