SAS® Cloud Analytic Services: CASL Programmer's Guide

2020.1 - 2021.1.5*

* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

SAS® Documentation
September 17, 2021

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2020. *SAS® Cloud Analytic Services: CASL Programmer's Guide*. Cary, NC: SAS Institute Inc.

SAS® Cloud Analytic Services: CASL Programmer's Guide

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

September 2021

v_001-P1:caslpg

Contents

1

Introduction to CASL Programming

About the CAS Language

CASL is a language specification used by the SAS client and other clients to interact with SAS Cloud Analytic Services (CAS).

Here are characteristics of the CAS language (CASL):

- CASL is a statement-based language.
- The language is case insensitive.
- CASL is a scripting language with the following strengths:
  - running actions
  - working with results
  - developing analytic pipelines
  - running code in CAS with user-defined actions
- Statements can include keywords such as the names of actions and functions.
- Statements can include expressions.
- Statements are terminated with a semicolon (;).
- A single PROC CAS step can contain several CASL programs.

Here are some uses for CASL:

- develop analytic pipelines

- use actions to submit requests to the CAS server to do work and then return the results
- evaluate and manipulate the results returned by an action
- create the arguments to an action
- create user-defined actions and functions

Running CASL Programs

To use the CASL language with SAS, you need the following:

- A CAS session. The CAS statement connects to an existing session on the server or starts a new session on the server. For more information about sessions, see "Sessions" in *SAS Cloud Analytic Services: Fundamentals*.

  TIP If you are not submitting CAS actions, then you do not need a CAS session.

- The CAS procedure. PROC CAS enables SAS to interpret the CASL programming statements that interact with the server.

You can submit CASL programs in the following ways:

- Through the SAS Windowing environment or SAS Studio using the CAS procedure.
- On the CAS server using server-side processing with user-defined actions. This enables you to run CASL programs from SAS, Python, Lua, or R. For more information about server-side processing, see Chapter 13, "Writing User-Defined Actions," on page 105.

2

CASL Data Types

What Are Data Types?

A data type is an attribute of every CASL variable. The data type is the characteristic that identifies a variable's value as a character string, an integer, and so on.

The following table lists the set of data types that are supported by CASL.

Table 2.1 *Frequently Used Data Types*

Data Type	Description
ARRAY	a list of values that are accessed by position.
BOOLEAN	a data type that permits only two values, True and False.

Data Type	Description
DICTIONARY	a list of key-value pairs that are accessed by key name.
DOUBLE	stores a signed, approximate, 64-bit double-precision, floating-point number. The DOUBLE data type stores numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.
INT32	a 32-bit signed, exact whole number, with a precision of 10 digits. The range of an INT32 is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values. Fractional portions are discarded.
INT64	a 64-bit signed, exact whole number, with a precision of 19 digits. The range of an INT64 is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values. Fractional portions are discarded.
STRING	a UTF-8 encoded sequence of characters.
TABLE	a result table. This is a two-dimensional composite data type with rows and columns.
VARCHAR	a varying-length UTF-8 encoded sequence of characters.

The following data types can be used by actions that run on the server, but are rarely used with CASL programming. There is no additional information in this document related to these data types.

Table 2.2 *Rarely Used Data Types*

Data Type	Description
BINARY	a fixed-length binary opaque data type used with in-memory tables in CAS. A SAS item store is also a binary. A SAS item store is used by SAS software to organize binary data.
DATE	a calendar date.
DATETIME	a data type that combines a calendar date with a time of day.
TIME	a data type that represents the time of day.
VARBINARY	a varying-length binary opaque data type used with in-memory tables in CAS. This data type is used to store image data, audio, documents, and other unstructured data.

Array

A CASL array is one of the two list data types. Arrays are most useful with CASL programming for grouping a series of strings or numbers in a variable and then using the variable as a parameter to a CAS action.

A CASL array can store any data type and multi-dimensional arrays are supported. This document focuses on simple single-dimensional arrays with a single data type for working with CAS actions.

Example Code 2.1 *Simple Array Assignment*

```
colNames = {"make", "model", "type"};
percentiles = {5, 25, 50, 75, 95};

name = colNames[2];

do v over colNames;
   print v;
end;
```

The PRINT statement shows the following in the SAS log:

```
make
model
type
```

Boolean

Overview

The Boolean data type is used to represent the values True or False.

Examples

Example Code 2.2 *Assign a Boolean Value to a Variable*

```
x = TRUE;
```

Example Code 2.3 *Assign a Value Based on a Logical Comparison*

```
one = 1;
```

```
t = one == 1;
f = one > 2;

print "T is: " t;
print "F is: " f;
run;
```

The PRINT statement shows the following in the SAS log:

```
T is: TRUE
F is: FALSE
```

Character

Overview

CASL has two character data types:

STRING
: a UTF-8 encoded sequence of characters. Character values that you assign to CASL variables in a program use this data type.

VARCHAR
: a varying-length UTF-8 encoded sequence of characters. This data type is used with CAS for in-memory tables on the server. This data type is used very rarely for CASL variables.

> **TIP** The VARCHAR data type is used in in-memory tables on the CAS server very frequently. However, when data is read from an in-memory table into a CASL variable, such as the result of an action, the STRING data type is assigned.

> **IMPORTANT** For VARBINARY data types, CASL truncates data to 64 characters for display only. However, CAS table data is not truncated.

Examples

Example Code 2.4 *Assign a String to a Variable*

```
name = 'Joe';                          /* 1 */
translate = '翻訳する';                /* 2 */
```

```
   describe name translate;                    /* 3 */
run;
```

1 The character value, Joe, is assigned to the variable that is named Name.

2 The character value for the word translate in the Japanese language is assigned to the variable that is named Translate.

3 The DESCRIBE statement is used to print the data types for the two variables to the SAS log.

The PRINT statement shows the following in the SAS log:

```
string;
string;
```

Example Code 2.5 *Replacing a String Variable*

```
   three = "Three";                            /* 1 */
   print "Variable three has a length of: " length(three);
run;

   three = "Thirteen";                         /* 2 */
   print "After assigning a new value, the length is: " length(three);
run;
```

1 A string value is assigned to the variable that is named Three.

2 When a different value is assigned to Three, CASL replaces the variable rather than allocating additional space for the increased length.

The PRINT statement shows the following in the SAS log:

```
Variable three has a length of: 5

After assigning a new value, the length is: 8
```

Dictionary

A CASL dictionary is one of the two list data types. A dictionary is an unordered list that is accessed by key.

A simple example of a CASL dictionary follows. The variable is named Tbl and has two keys, Name and Caslib.

Example Code 2.6 *Simple CASL Dictionary with Two Keys, Long Form*

```
   tbl.name = "iris";
   tbl.caslib = "casuser";

   describe tbl;
run;
```

The following code is equivalent to the code above.

Example Code 2.7 *Simple CASL Dictionary with Two Keys, Short Form*

```
tbl={name="iris", caslib="casuser"};

   describe tbl;
```

The DESCRIBE statement shows the following in the SAS log:

```
dictionary ( 2 entries, 2 used);
 [name] string;
 [caslib] string;
```

- Dictionaries are useful for organizing complex, nested parameters for a CAS action in a single variable. Then, you can use the variable with the CAS action to supply the nested parameters.
- The results of an action are a dictionary. Dictionaries contain values. The most common value in the dictionary is the result table. It is important to understand how to identify and specify dictionary keys in order work with CAS action results.
- A CASL dictionary can store any data type. This document focuses on simple uses that can help you simplify your CASL programming.

Numeric

Overview

CASL supports three numeric data types:

INT32
: signed 32-bit integer

INT64
: signed 64-bit integer

DOUBLE
: signed double-precision floating point number

- When a numeric operation uses a double, the result is a double.
- The INT32 data type can be assigned to a column in a result table. When whole numbers are assigned to a CASL variable, INT64 is used.

A missing numeric value is smaller than any other numeric value, and missing numeric values have their own sort order. Missing values are covered in another part of this document.

Examples

Example Code 2.8 *Assign a 64-Bit Integer Value to a Variable*

```
ten = 10;                                   /* 1 */
ten = 10L;

tenhex = 0Ax;                               /* 2 */

twobin = 10b;                               /* 3 */

eightK = 8k;                                /* 4 */

mega = 1m; /* 1024**2 */

giga = 1g; /* 1024**3 */
```

1 By default, the INT64 data type is used for whole numbers. In the second line, the L suffix is used to set the INT64 data type explicitly.

2 X is a supported suffix for hexadecimal radix.

3 B is a supported suffix for binary radix.

4 K is a supported suffix for kilobytes. The specified value is multiplied by 1024. M and G are also supported suffixes.

Example Code 2.9 *Assign a Double-Precision Value to a Variable*

```
d = 10.0;                                   /* 1 */

kilo = 1.0k;                                /* 2 */
mega = 1.0m;
giga = 1.0g;

half = 1 / 2;
```

1 Include a decimal point to ensure that a whole number is assigned a DOUBLE data type.

2 K is a supported suffix for DOUBLE. The value is multiplied by 1024. M and G are similar and multiply the value by 1024^2 and 1024^3 respectively.

Arithmetic operations with at least one variable that is a DOUBLE result in a DOUBLE for the result.

Example Code 2.10 *Assign a Value Based on a Numeric Operation*

```
ten = 10;
fivePointFive = 5.5;
diff = ten - fivePointFive;
```

Result Table

A result table is a two-dimensional data representation. It is a common data type that is created as a result of an action.

- Each column has a name, a data type, an optional label, and an optional format.
- A result table can have attributes to describe the contents of the table.
- A result table is represented as a two dimensional array. The first dimension references the row number. The second dimension is the column number or column name.
- When CASL is used with SAS, SAS Studio, SAS Enterprise Guide, and so on, the output delivery service is used to display result tables.

3

CASL Variables

Overview

A CASL variable has a name, a scope, and a data type.

- Variable names can start with alphabetic character, _ (underscore), or $. The name can contain numbers, $, or _ (underscore).
- Variables can have either a global or local scope.
- The data type of a variable is determined by the type of value assigned to it.
  - If the data type is ARRAY, DICTIONARY, STRING, TABLE, VARBINARY, or VARCHAR, the value is passed by reference to the memory location containing the actual data. This data is separate (or outside) the function. This is advantageous because the data does not need to be copied.
  - If the data type is BOOLEAN, INT32, INT64, DATE, DATETIME, or TIME, the size of the data is small, and the value's scope is contained within the function. Instead of passing a reference to the value, the value itself is passed to a function.

You can use CASL variables in these ways:

- as arguments to an action.
- in an expression. The return value can be a value, an array, or a dictionary. Values that are STRING, TABLE, ITEMS, or VARBINARY, are passed by a reference.
- as an alias to a function. You can assign a function to a variable.

Basic Syntax

Use an assignment statement to evaluate an expression and store the result in a variable. The basic syntax for creating a variable is the following:

variable = expression

This assignment statement defines the target variable as a varchar variable with the value “Jane Smith”.

```
Name = 'Jane Smith';
```

This assignment statement defines the target variable as an integer with the value of “88”.

```
 xx.y = 88;
```

This target statement assigns the result of the MIN function to the variable z.

```
z = min(y, 70);
```

If a variable already exists, the new assignment replaces the variable and the old value is overwritten. Use the DESCRIBE statement to show a variable’s internal structure. For more information, see Chapter 6, “Using the DESCRIBE Statement,” on page 35.

Variable Scope

Variables have local or global scope. Local variables exist only within the function where they were created. Global variables can be accessed and used anywhere within a CASL program. You can use the LOCAL statement to explicitly create a variable that is local to a function, and the GLOBAL statement to explicitly create a global variable.

CAUTION

If you do not use the LOCAL statement to create a local variable in a function, the variable is implicitly global. If a global variable of the same name exists, the global variable is referenced. It is a best practice to use the LOCAL statement when creating variables inside a function.

- Global variables can be accessed and used anywhere within a CASL program.
- To explicitly create a global variable, use the GLOBAL statement.
- A variable is implicitly global if it is created outside of a function.
- A variable is implicitly global if it is created inside of a function, but is not a control loop variable or created by the LOCAL statement
- This program creates a global variable called Rental.

```
proc cas;
   function myFunction( y );
      global x;              1
      print "NOTE: When myFunction begins execution, x=" x " and y=" y;
      x=y;
   end func;
run;
   x=5;                      2
   print "NOTE: Before myFunction execution, x=" x;
   myFunction(2);
   print "NOTE: After myFunction execution, x=" x;
run;
quit;
```

1 The GLOBAL statement creates a variable named x that has global scope.

2 The x variable created outside of the function is also global.

- Starting with SAS Viya 2021.1.5, you can use the LOCAL statement to explicitly create a variable that is local to the function where it is created. Variables created using the LOCAL statement exist only within that function.
- Starting with SAS Viya 2021.1.5, loop iteration variables used within a function implicitly have local scope and exist only within that function.
- Local variables persist until the function returns.
- This program creates a local variable:

```
proc cas;
   function myFunction( y );
      local x;               1
      print "NOTE: When myFunction begins execution, x=" x " and y=" y;
      x=y;
      print "NOTE: After the myFunction assignment statement, x=" x "
and y=" y;
   end func;
run;
   x=5;                              2
   print "NOTE: Before myFunction execution, x=" x;
   myFunction(2);
   print "NOTE: After myFunction execution, x=" x;
run;
quit;
```

1 The LOCAL statement creates a variable named x that has local scope.

2 The x variable created outside of the function is global.

- Starting with SAS Viya 2021.1.5, loop iteration variables used within a function implicitly have local scope and exist only within that function.

See Also

- “Assignment Statement” in *SAS Cloud Analytic Services: CASL Reference*

- “GLOBAL Statement” in *SAS Cloud Analytic Services: CASL Reference*

4

CASL Expressions

Definitions for CASL Expressions

The following definitions apply to CASL expressions:

expression
: is a sequence of operands and operators that form a set of instructions that are performed to produce a resulting value. You use expressions in CASL program statements to create variables, assign values, calculate new values, transform variables, and perform conditional processing. Expressions can produce a value that has any data type.

operands
: are constants or variables that can be numeric or character. The result of an expression can be an operand.

operators
: Operators are the symbols that represent a calculation, comparison, or concatenation of operands.

Examples of Expressions

Here are some examples of CASL expressions:

- `5`
- `x+1`
- `years*12`
- `min(y, 70)`
- `nmissVar + col.Column`
- `(trim(row._VARNAME_) != targetVar)`
- `exp( lgamma(days+1) - lgamma(days-number+1) - number*log(days))`

Operators in Expressions

Overview

A CASL operator is a symbol that represents a comparison, arithmetic calculation, or logical operator. Expressions are evaluated to produce a value by using operators.

The following data types have their own operators or override the operators in this section.

- Result Table Operators on page 87
- Array Operators on page 42
- Dictionaries on page 53

The operation that an operator performs depends on the data type of the operands. Having the operation of an operator change based on the data types of the operands is called overloading. Overloading simplifies the syntax for common operations and provides more intuitive programming. Here are some examples:

- The plus sign arithmetically adds two numeric operands while the plus sign unions two array operands.
- The minus sign arithmetically subtracts two numeric operands while the minus sign also computes the intersection of two lists.

- Multiplying a table and a list changes the labels in the table to the list.

Arithmetic Operators

Arithmetic operators indicate that an arithmetic calculation is being performed. The following table shows the precedence used to determine the order in which arithmetic expressions for the addition, subtraction, multiplication, and division operators are evaluated. 1 is the highest precedence and 4 is the lowest.

Table 4.1 *Arithmetic Operators*

Precedence	Symbol	Definition	Example
1	+ (unary) - (unary)	Indicates a positive or negative number.	The following express resolves to -5: `x = -5;`
2	**	Power	`x**2`
3	/ *	Division Multiplication	`var/2` `6.5*salary`
4	+ -	Addition Subtraction	`num+4` `sale-discount`

Note: The asterisk (*) is always necessary to indicate multiplication; 2Y and 2(Y) are not valid expressions.

Comparison Operators

Comparison operators set up a comparison, operation, or calculation with two variables: constants, or expressions. Comparison operators can be expressed as symbols or they can be expressed with their mnemonic equivalents, which are shown in the following table:

Table 4.2 *Comparison Operators*

Symbol	Mnemonic Equivalent	Definition	Example
>	GT	Greater than	rate > 10
<	LT	Less than	rate < 40

Symbol	Mnemonic Equivalent	Definition	Example
=	EQ	Equal to	rate = 15
==	EQ	Equal to	rate == 15
!=	NE	Not equal to	rate != 25
>=	GE	Greater than or equal to	rate>=10
<=	LE	Less than or equal to	rate<=40
>:	GTC	Truncated greater than conditional	rate>:10
<:	LTC	Truncated less than conditional	rate<:40
=:	EQC	Truncated equal to conditional	"abc" =: "abcdef";
>=:	GEC	Truncated greater than conditional	"abc" >=: "abcdef";
<=:	LEC	Truncated less than conditional	"abc" <=: "abcdef";

- The precedence of comparison operators is less than arithmetic operators.
- Character operands are compared character by character from left to right. Character order depends on the collating sequence, usually ASCII or EBCDIC, used by your computer. For example, in the EBCDIC and ASCII collating sequences, *R* is greater than *G*. Therefore, this expression is True:

  ```
  'Raymond' > 'Gibson'
  ```

- Two-character values of unequal length are compared as if blanks were attached to the end of the shorter value before the comparison is made. A blank, or missing character value, is smaller than any other printable character value. For example, because . is less than *h*, this expression is true:

  ```
  'C.Mills' < 'Charles Mills'
  ```

  Since trailing blanks are ignored in comparison, `'dog '` is equivalent to `'dog'`. However, because blanks at the beginning and in the middle of a character value are significant to SAS, `' dog'` is not equivalent to `'dog'`.

- You can compare only a specified prefix of a character expression by using a colon (:) after the comparison operator. In the following example, the colon modifier after the equal sign tells SAS to look at only the first character of values of the variable LastName and to select the observations with names beginning with the letter F:

  ```
  if lastname=:'F';
  ```

  The printable characters are greater than blanks. Both of the following statements select observations with values of LastName that are greater than or equal to the letter F:

- `if lastname>='F';`
- `if lastname>=:'F';`

Boolean (Logical) Operators

Boolean operators, also called logical operators, are used to connect and define the relationship between expressions to link sequences of comparisons together. The logical operators and their precedence are shown in the following table. 1 is the highest precedence and 5 is the lowest.

Table 4.3 *Boolean Operators*

Precedence	Symbol	Mnemonic Equivalent	Definition	Example
1	none	BETWEEN	Selects observations by searching for a specified value within the values of a character variable	10 BETWEEN 1 AND 100
1	none	CONTAINS	Selects observations by searching for a specified set of characters within the values of a character variable	If company is "SF Bay Trading", then the following is TRUE: company CONTAINS "Bay"

1	IN	IN	IN	`x=5;`
				`y = x in (1:10);`
				`print y;`

1	LIKE	LIKE	The LIKE operator selects observations by comparing the values of a character variable to a specified pattern, which is referred to as pattern matching. The LIKE operator is case sensitive. There are two special characters available for specifying a pattern:	If lastName is "Malik", then the following are TRUE: ■ lastName LIKE "Ma_ik" ■ lastName LIKE "M%"

Precedence	Symbol	Mnemonic Equivalent	Definition	Example
			■ percent sign (%) ■ underscore (_)	
2	><	MIN	Returns the lower of the values	Invoice><MSRP
2	<>	MAX	Returns the higher of the values	Invoice<>MSRP
3	! NE ~ ^	NOT	Logical NOT	!(gradesA>gradesB)
4	&	AND	Logical AND	(gradesA>gradesB&gradesC>gradesD)
5	|	OR	Logical OR	(gradesA>gradesB|gradesC>gradesD)

Note: If you compare a zero-length character value with any other character value in either an IN: comparison or an EQ: comparison, the two-character values are not considered equal. The result always evaluates to 0, or false.

Other Operators

Table 4.4 *Other Operators*

Symbol	Definition	Example
(	Start a sub expression	(trim(row._VARNAME_) != targetVar)
)	End a sub expression	(trim(row._VARNAME_) != targetVar)
`expr ? <true-expr> : <false-expr>;`	Selects a value depending on whether the expression is true.	x=(5>3) ? 4 : 8 If the value of (5>3) is true, then the value is 4, otherwise, the value is 8. This example evaluates to 4.
||	Concatenates character values.	'JOHN '||'SMITH'; The expression evaluates to JOHN SMITH.

Order of Precedence

The following table shows all of the operators and their order of precedence. The highest order of precedence is 1 and the lowest order of precedence is 13.

Table 4.5 *Operators and Their Order of Precedence in CASL Expressions*

Precedence	Operator
1	( )
2	+ (unary) – (unary)
3	**
4	* /
5	+ –
6	>, GT <, LT =, EQ = =, EQ !=, NE >=, GE <=, LE >:, GTC <:, LTC =:, EQC >=:, GEC <=:, EC
7	LIKE IN BETWEEN CONTAINS, ?

8	>< , MIN <>, MAX
9	NOT (~, !, NE, ^)
10	&, AND
11	|, OR
12	?: *expression* ? <*true-expression*> : <*false-expression*>;
13	| |

Type Conversions

Overview

Operands in an expression must be of the same general data type (numeric, character, binary, or date/time) in order for CASL to evaluate the expression. When it is necessary, CASL converts an operand's data type to another data type, depending on the operands and operators in the expression. This process is called type conversion.

Numeric Type Conversion

- In an arithmetic expression, the DOUBLE data type has a higher precedence than INT64. In the addition of a DOUBLE and an INT64, the expression evaluates to a DOUBLE. For example, in the following code, the expression evaluates to a DOUBLE:

```
proc cas;
y=5.0 + 5;
describe y;
run;
```

- The addition operator (+) operates on numeric data types. If an integer is in a numeric operation with a STRING, the STRING is converted to a value and then that value determines the resulting type. For example, in the addition of the character string ‘5’ and the numeric integer 5, the STRING data type for the operand ‘5’ is converted to an INT64 data type before the evaluation takes place.

```
proc cas;
```

```
y="5" + 5;
describe y;
run;
```

The log shows the following message: .

```
NOTE: Character value '5' has been converted to numeric
int64_t;
```

Comparison Type Conversion

Boolean operators (AND, OR, NOT) operate on TRUE or FALSE values. When the type of a Boolean operator's operand is not Boolean, the value is converted to Boolean (TRUE or FALSE). For example:

- When the type is numeric nonzero, values are converted to TRUE and zero is converted to FALSE.
- When the type is a character value, the character value is converted to a numeric value, then the rules for numeric conversion are followed.

Casting Data Types

Basic Syntax

The (cast) operator is used to convert a value explicitly from one data type to another data type.

(*data-type*) *variable-or-expression*

For example, the data type of the value `1` is INT64. You can explicitly change the data type to STRING. In the following example, the expression that is highlighted evaluates to a string.

```
proc cas;
   x = "Product #" || (string)1;
   print x;
run;
```

The PRINT statement shows the following in the SAS log:

```
Product #1
```

The following are valid casting types:

- (INT64)
- (DOUBLE)
- (STRING)

- (BOOLEAN)

Casting is permitted anywhere expressions are permitted.

See Also

- Chapter 2, “CASL Data Types,” on page 3
- Chapter 5, “CASL Functions,” on page 25
- Chapter 7, “CASL Arrays,” on page 41
- Chapter 8, “CASL Dictionaries,” on page 53

5

CASL Functions

Overview

A function is a component of the CASL programming language that can accept arguments, perform a computation or other operation, and return a value. The value that is returned can be used in an assignment statement or elsewhere in expressions. The CASL language provides built-in functions that provide functionality that is unique to CASL, and functions that provide functionality found throughout SAS software. You can also create your own user-defined functions. Here are some examples of CASL functions:

Function type	Example
built-in	`readpath ("/u/sasdemo/ds/samplecode.sas");`
common	`datetime();`
user-defined	`SharedBday(365,n);`

Functions Supplied by SAS

Overview

SAS provides two types of supplied functions, built-in functions and common functions.

Built-in functions have functionality that is unique to CASL. These functions enable you to perform operations on your result tables, arrays, and dictionaries, and provide run-time support for your CASL programs. These CASL functions cannot be replaced with user-defined functions. For a list of built-in functions, see "CASL Built-In Functions" in *SAS Cloud Analytic Services: CASL Reference*.

Common functions provide functionality that is common to other SAS functions. When used in a CASL program, SAS functions take a CASL value and a CASL value is returned. You can replace these functions with user-defined functions. For a list of common functions that can be used in CASL programs, see "Common Functions" in *SAS Cloud Analytic Services: CASL Reference*.

Examples

Read in Code Saved Externally

The following example uses the READPATH function to read in a file that contains a DATA step program and store the code in a CASL variable. You can then use the variable that contains the code as input to an action. Before you run this example, you must load the Cars data set into CAS as an in-memory table. See "Load a Client-Side File " on page 58.

Save the following DATA step code in a file named carsCode.sas. This is the program that is read into CAS and stored as a variable.

```
data bigCars;
   set cars; by make type;
   keep make type weight;
   if weight < 5000 then delete;
   run;
```

The runCode action runs DATA step code in CAS. You can store your DATA step code in a CASL variable, and then use the CASL variable as input to the code parameter.

```
proc cas;
   carsCode = readpath ("file-path/carsCode.sas");   /*1*/
   datastep.runCode /                                /*2*/
```

```
      code = carsCode;                                    /* 3 */
run;
   table.fetch / to = 5                                   /* 4 */
      table = "bigCars"
      sortby = {
         {name = "weight", order = "descending"}
               };
run;
```

1 The READPATH function reads in the contents of the file carsCode.sas as a text string. The string is stored in the variable carsCode.

2 The runCode action runs the DATA step in CAS.

3 The code parameter specifies the variable that contains the DATA step code.

4 The fetch action displays the results table. The sortBy parameter sorts the values for Weight in descending order.

Output 5.1 *Results: Fetched Rows from the BigCars Table*

Selected Rows from Table BIGCARS			
Index	Make	Type	Weight
1	Ford	SUV	7190
2	Hummer	SUV	6400
3	GMC	SUV	6133
4	Lincoln	SUV	5969
5	Cadillac	Truck	5879

List Functions

You can use the FUNCTIONLIST statement to print a list all of the built-in functions to the SAS log.

```
proc cas;
   functionlist;
run;
```

Output 5.2 *Partial Listing of Built-in Functions Available in CASL*

```
NOTE: CASL Built-In functions
NOTE: actionstatus    : Return TRUE if the session is active with an action
NOTE: add_table_attr  : Add attributes to a result table
NOTE: addbygroup      : Creates a new result table from a BY-group table
NOTE: addrow          : Add a row to a result table
NOTE: addunique       : Add a value to a list if the value does not exist in a list
NOTE: cancelaction    : Cancels the action running on the given session
NOTE: cancelactions   : Cancel the actions on the given sessions
NOTE: clear           : Clear the given value
NOTE: codetostatus    : Convert the action exit status into a dictionary
NOTE: combine_tables  : Create a new result table that has the name of the first table,
                        and contains all rows from all tables
NOTE: dictionary      : Search for a value in a dictionary given the key
NOTE: dim             : Retrieve the dimensions of a variable
```

Use the FNC statement to list the common functions.

```
proc cas;
   fnc;
run;
```

Output 5.3 *Partial Listing of Common Functions Available in CASL*

```
NOTE: abs           (math)       Returns the absolute value of a numeric value.
NOTE: airy          (math)       The AIRY function returns the value of the airy function
NOTE:                            (Abramowitz and Stegun 1964; Amos, Daniel and Weston 1977)
NOTE:                            (See References). It is the solution of the differential equation.
NOTE: anyalnum      (char)       Searches a character string for an alphanumeric character,
NOTE:                            and returns the first position at which the character is found.
NOTE: anyalpha      (char)       Searches a character string for an alphabetic character,
NOTE:                            and returns the first position at which the character is
NOTE:                            found
NOTE: anycntrl      (char)       Searches a character string for a control character, and
NOTE:                            returns the first position at which that character is found.
NOTE: anydigit      (char)       Searches a character string for a digit, and returns the
NOTE:                            first position at which the digit is found.
NOTE: anyfirst      (char)       Searches a character string for a character that is valid
NOTE:                            as the first character in a SAS variable name
NOTE: anygraph      (char)       Searches a character string for a graphical character, and
NOTE:                            returns the first position at which that character is
NOTE:                            found.
NOTE: anylower      (char)       Searches a character string for a lowercase letter, and
NOTE:                            returns the first position at which the letter is found.
NOTE: anyname       (char)       Searches a character string for a character that is valid
NOTE:                            in a SAS variable name under VALIDVARNAME=V7, and returns
NOTE:                            the first position at which that character is found.
NOTE: anyprint      (char)       Searches a character string for a printable character, and
NOTE:                            returns the first position at which that character is
NOTE:                            found.
```

User-Defined Functions

Overview

You can use the FUNCTION statement to define your own CASL functions.

- User-defined functions persist only during the current PROC CAS step.
- Variables created during the execution of a user-defined function are local and are dropped when the function returns.
- The CASL GLOBAL statement can be used to create a global variable from within a function.
- You do not specify the parameter's data type.
- The parameters take the type of the value sent to the function.
- Multiple RETURN statements can be used to return a value at any time.

- User-defined functions can be used in place of any of the common functions found in the section "Common Functions" in *SAS Cloud Analytic Services: CASL Reference*.

Basic Syntax

The basic syntax for the FUNCTION statement is the following:

FUNCTION *function- name* (<*parameter-1*, *parameter-2*>);
…<*CASL-statements*>…;
RETURN (*expression*);
END;

CASLstore

CASLstore is a collection of built-in CASL functions that enable you to store CASL functions and snippets on the server. The CASLstore facility consists of three functions: CASLSTORE Function, DEFAULT_CASLSTORE Function, UPLOAD_CASLSTORE Function. CASLstore works by saving your CASL functions to a CAS table on the server. Once the table is saved to the server, you must specify that table, or a list of tables, for CAS to search for functions. After specifying the location of the table using the CASLstore functions, CAS will dynamically download, compile, and execute your user-defined functions as needed by your CASL program.

Examples

Create User-Defined Functions

The following example creates a function that converts temperatures from Fahrenheit to Celsius, and then calls the function to convert a list of temperatures. Variables created during the execution of this function are local, and they disappear when the function returns. The GLOBAL statement can be used to create a global variable from within a function.

Example Code 5.1 *Create a Function to Convert Fahrenheit to Celsius*

```
proc cas;
   function FtoC(temp);              /* 1 */
      Celsius = (5/9*(temp-32));     /* 2 */
      return(Celsius);               /* 3 */
   end func;

   tempF = {30 35 31 29};
   do n over tempF;                  /* 4 */
```

```
      Celsius = FtoC(n);              /* 5 */
      print put(Celsius, best6.2);    /* 6 */
   end;
run;
```

1 The FUNCTION statement creates a new function. In this example, the function is named FtoC and has one argument named temp.

2 The assignment statement creates a variable named Celsius that holds the converted temperature.

3 The RETURN statement returns a value from the current function.

4 Iterate over the list of values and call the FtoC function for the value of N.

5 Call the FtoC function and store the result in the variable Celsius.

6 The PRINT statement prints the result to the SAS log. The PUT function assigns the BEST6.2 format to the values.

Example Code 5.1 *Values Returned by the Function FtoC*

```
-1.111
1.6667
-0.556
-1.667
```

The following example creates a function to determine the probability for two people in a room to have their birthday on the same day. The program accepts as parameters the number of people and the number of birthdays in a year.

Example Code 5.2 *Create a Function to Calculate Probability*

```
proc cas;
   function SharedBday(days,
number);                                          /* 1 */
      p = exp( lgamma(days+1) - lgamma(days-number+1) -
number*log(days));    /* 2 */
      return (1-
p);                                                          /* 3 */
   end func;

   do n over {3 10 22 23 50
75};                                              /* 4 */
      p =
SharedBday(365,n);                                                  /*
5 */
      print "Chance at least 2 out of " put(n,best3.) "
             share the same birthday = "
put(p,best6.2);                       /* 6 */
   end;
run;
```

1 The FUNCTION statement creates a new function. In this example, the function is named SharedBday and it has two arguments, Days and Number.

2 Specify the calculation for determining the probability of shared birthdays. The Assignment statement creates the local variable P that contains the value of the calculation.

3 The RETURN statement returns a value from the current function.

4 The DO OVER statement iterates over the array.

5 Call the SharedBday function and store the result in the variable P.

6 The PRINT statement prints the result to the SAS log. The PUT function assigns the BEST6.2 format to the values.

Output 5.4 *Values Returned by the SharedBday Function*

```
Chance at least 2 out of   3 share the same birthday = 0.0082
Chance at least 2 out of  10 share the same birthday = 0.1169
Chance at least 2 out of  22 share the same birthday = 0.4757
Chance at least 2 out of  23 share the same birthday = 0.5073
Chance at least 2 out of  50 share the same birthday = 0.9704
Chance at least 2 out of  75 share the same birthday = 0.9997
```

List User-Defined Functions

You can use the FUNCTIONLIST statement to print a list of user-defined functions to the SAS log. The following example creates three user-defined functions, and then prints a list of them to the SAS log.

```
proc cas;
   function FtoC(temp);
      Celsius = (5/9*(temp-32));
      return(Celsius);
   end func;

   function SharedBday(days, number);
      p = exp( lgamma(days+1) - lgamma(days-number+1) -
number*log(days));
      return (1-p);
   end func;

   function factorial(x);
      if (x < 1.0) then return(x);
      else do;
         return exp(lgamma (x+1));
      end;
   end func;

functionlist user;
run;
```

Output 5.5 *Results of the FLIST Statement*

```
NOTE: User defined functions
NOTE: factorial( x);
NOTE: FtoC( temp);
NOTE: SharedBday( days, number);
```

Upload and Execute Stored User-Defined Functions Using CASLstore

The following example uses the CASLstore functions to save CASL functions to a CAS table. You can then execute the functions in a CASL program.

```
proc cas;
source
funcs;
    /* 1 */
   function factorial(x);
      if (x < 1.0) then return(x);
      else do;
         return exp(lgamma (x+1));
      end;
   end func;

   function SharedBday(feature,number);
      p = exp( lgamma(feature+1) - lgamma(feature-number+1) -
number*log(feature));
      return (1-p);
   end func;

   function FtoC(temp);
      Celsius = (5/9*(temp-32));
      return(Celsius);
   end func;
endsource;


upload_caslstore({caslib="casuser",
          /* 2 */
     name="store1", replace=true}, funcs);

run;

proc cas;
   cs = caslstore({caslib="casuser",
name="store1"});                                  /* 3 */

   x = factorial( 10);
   print "10! = " x;
   run;

   bday = SharedBday(365, 23);
   print "Chance for 23 people to share the
          same birthday = "  put(x,best7.4);
   run;

   tempF = {30 35 31 29};
   do n over tempF;
      Celsius = FtoC(n);
```

```
      print put(Celsius, best6.2);
   end;
run;
```

1. Use a source statement to define the Factorial, SharedBday, and FtoC functions.
2. Save the functions to the store1 CAS table in the caslib CASUSER using the UPLOAD_CASLSTORE function.
3. Specify the location of the CAS table containing your functions using the CASLSTORE function.

Output 5.6 *Values Returned by the Factorial, SharedBday, and FtoC Functions*

```
     x = factorial( 10);
     print "10! = " x;
     run;
10! = 3628800

     bday = SharedBday(365, 23);
     print "Chance for 23 people to have the same birthday = "  put(bday,best7.4);
     run;
Chance for 23 people to have the same birthday = 0.5073

     tempF = {30 35 31 29};
     do n over tempF;
        Celsius = FtoC(n);
        print put(Celsius, best6.2);
     end;
     run;
-1.111
1.6667
-0.556
-1.667
```

Key Ideas

- The argument types are unspecified, and take the type of the argument passed to the function. Values that have a data type of varbinary, dictionary, or array are specified by address. All other data types are specified by value. The return value can be any data type.
- A function can be defined with any number of parameters. However, if you try to pass more values to the function than are supported, the additional values are not accessible.
- You can use the FNC statement to list the functions available to CAS by name and category. Use the FUNCTIONLIST statement for a list of user-defined functions or built-in functions.
- User-defined functions persist only during the current PROC CAS step. You can store your user-defined functions in a text file and use the global SAS statement %INCLUDE to include the file within a PROC CAS step.

- Variables created during the execution of a function are local and are dropped when the function returns. The CASL GLOBAL statement can be used to create a global variable from within a function.
- You can assign a function to a variable. The variable then becomes an alias for the function.

See Also

- “Assignment Statement” in *SAS Cloud Analytic Services: CASL Reference*
- Chapter 6, “Using the DESCRIBE Statement,” on page 35
- “DO OVER Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “FUNCTION Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “GLOBAL Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “PRINT Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “PUT Function” in *SAS Cloud Analytic Services: CASL Reference*
- “RETURN Statement” in *SAS Cloud Analytic Services: CASL Reference*

6

Using the DESCRIBE Statement

Overview

The DESCRIBE statement enables you to view the structure and data type of CASL variables and expressions. When action results are saved to a variable, the DESCRIBE statement writes a description of the variable to the SAS log. The description shows the expanded arrays and dictionaries in order to view the structure of the variable.

When you use CASL with PROC CAS client side, the DESCRIBE statement writes to the SAS log. When you submit server-side CASL, the output is written to the journal for the language that you are using. For information about using server-side CASL, see Chapter 13, “Writing User-Defined Actions,” on page 105.

Basic Syntax

DESCRIBE *variable-or-expression*;

The following table shows the information displayed for different data types.

Data Type	Information Displayed by DESCRIBE statement
dictionary	data type number of entries in the dictionary number of entries used
array	data type number of entries in the dictionary number of entries used
result table	data type column formats column names column types number of columns number of rows table name
character	data type
Boolean	data type
dates and time	data type
numeric	data type
varbinary	data type
item	data type

Examples

Select a Result Table for Printing

The following example generates a matrix of Pearson product-moment correlation coefficients for the data in table Iris. The Correlation action creates two result tables, CorrSimple and Correlation. The PRINT statement selects the CorrSimple table to be printed to the output destination.

Note: Before you can work with an in-memory table in CAS, you must download the file Iris.csv and load it into CAS. For the complete code for accessing the Iris data set, see "Save and Drop an In-Memory Table " on page 61.

```
proc cas;
   simple.correlation result=corrRes /                          /* 1 */
      inputs={"Sepal Length", "Sepal Width"}
      pairWithInput={"Petal Length", "Petal Width"}
      table={name="iris",
             where="Species eq 'setosa'"
      };
run;

describe corrRes;                                               /* 2 */
run;

print corrRes.CorrSimple;                                       /* 3 */
run;
quit;
```

1 The Correlation action computes Pearson product-moment correlations.

2 The DESCRIBE statement prints the structure of the variable corrRes to the SAS log. You can use this information to select one of the tables for printing.

3 The PRINT statement selects the table CorrSimple and displays it to the open ODS output destination.

The following output shows that the corrRes variable is a dictionary that contains two tables. The table names are CorrSimple and Correlation. You can use the PRINT statement to select one of the tables to be printed.

Output 6.1 *Data Structure of the CorrRes Variable*

```
dictionary ( 2 entries, 2 used);
[CorrSimple] Table ( [4] Rows [7] columns
Column Names:
 [1] Variable          [Analysis Variable] (char)
 [2] N                 [N                ] (double) [BEST10.]
 [3] Mean              [Mean             ] (double) [D8.4]
 [4] Sum               [Sum              ] (double) [BEST10.]
 [5] StdDev            [Std Dev          ] (double) [D8.4]
 [6] Minimum           [Min              ] (double) [D8.4]
 [7] Maximum           [Max              ] (double) [D8.4]
 [Correlation] Table ( [2] Rows [3] columns
Column Names:
 [1] Variable          [                 ] (char)
 [2] Sepal Length      [                 ] (double) [D8.4]
 [3] Sepal Width       [                 ] (double) [D8.4]
1021  describe corrRes;
```

Output 6.2 *Results: Printed Contents of the CorrSimple Table*

CorrSimple: Results from simple.correlation

Summary Statistics in Correlation Analysis for IRIS						
Analysis Variable	N	Mean	Sum	Std Dev	Min	Max
Petal Length	150	3.7587	563.8	1.7644	1.0000	6.9000
Petal Width	150	1.1987	179.8	0.7632	0.1000	2.5000
Sepal Length	150	5.8433	876.5	0.8281	4.3000	7.9000
Sepal Width	150	3.0540	458.1	0.4336	2.0000	4.4000

View the Data Structure and Values of a Variable

The following example uses the Summary action to group summary statistics in the table Cars by Type. The summary is performed on Mpg_City and Mpg_Highway.

Note: Before you can work with an in-memory table in CAS, you must download the file Cars.csv and load it into CAS. For the complete code for accessing the Cars data set, see "Load a Client-Side File " on page 58.

Example Code 6.1 *Create the Summary Results*

```
proc cas;
   simple.summary
result=bygrps /                                          /*1*/
      inputs={"mpg_city" "mpg_highway"}
      subset={"min","max","sum","mean", "nmiss"}
      table={name="cars",
            groupBy={{name="Type"}}};

   describe
bygrps;                                                    /*2*/
   run;
   print
bygrps;                                                     /*3*/
   run;
```

1 The summary action computes the statistics for Mpg_City and Mpg_Highway by type in table cars and assign the results to the variable Bygrps.

2 The DESCRIBE statement prints the structure of the variable Bygrps to the SAS log.

3 The PRINT statement prints the variable Bygrps. Because Bygrps is a result table, it is printed to the open ODS output destination.

The following output shows information about the first three group-by tables.

Output 6.3 *Data Structure of the Bygrps Variable*

```
dictionary ( 7 entries, 7 used);
  [ByGroupInfo] Table ( [6] Rows [3] columns
 Column Names:
  [1] Type              [                ] (varchar)
  [2] Type_f            [                ] (char)
  [3] _key_             [                ] (varchar)
  [ByGroup1.Summary] Table[Summary] ( [2] Rows [6] columns
 Column Names:
  [1] Column            [Analysis Variable] (char)
  [2] Min               [Minimum         ] (double) [D8.4]
  [3] Max               [Maximum         ] (double) [D8.4]
  [4] NMiss             [N Miss          ] (double) [BEST10.]
  [5] Mean              [Mean            ] (double) [D8.4]
  [6] Sum               [Sum             ] (double) [BEST10.]
  [ByGroup2.Summary] Table[Summary] ( [2] Rows [6] columns
 Column Names:
  [1] Column            [Analysis Variable] (char)
  [2] Min               [Minimum         ] (double) [D8.4]
  [3] Max               [Maximum         ] (double) [D8.4]
  [4] NMiss             [N Miss          ] (double) [BEST10.]
  [5] Mean              [Mean            ] (double) [D8.4]
  [6] Sum               [Sum             ] (double) [BEST10.]
  [ByGroup3.Summary] Table[Summary] ( [2] Rows [6] columns
 Column Names:
  [1] Column            [Analysis Variable] (char)
  [2] Min               [Minimum         ] (double) [D8.4]
  [3] Max               [Maximum         ] (double) [D8.4]
  [4] NMiss             [N Miss          ] (double) [BEST10.]
  [5] Mean              [Mean            ] (double) [D8.4]
  [6] Sum               [Sum             ] (double) [BEST10.]
```

The following output shows the first two group-by tables.

Output 6.4 *Results: Printed Bygrps Tables*

bygrps: Results from simple.summary

Type=Hybrid

Descriptive Statistics for CARS					
Column	Minimum	Maximum	Sum	Mean	N Miss
MPG_City	46.0000	60.0000	165	55.0000	0
MPG_Highway	51.0000	66.0000	168	56.0000	0

bygrps: Results from simple.summary

Type=Sedan

Descriptive Statistics for CARS					
Column	Minimum	Maximum	Sum	Mean	N Miss
MPG_City	12.0000	38.0000	5524	21.0840	0
MPG_Highway	17.0000	46.0000	7501	28.6298	0

Example Code 6.2 *Create a List of Car Types with Mileage Greater Than Twenty*

```
Mileage= {};                                                    /* 1 */
do i = 2 to dim(bygrps);
   if ( bygrps[i][1,"Mean"] >20) then
      Mileage = Mileage + bygrps[i].attrs.ByGroup;
end;

describe Mileage;                                               /* 2 */
run;

print Mileage;                                                  /
* 3 */
run;
```

1 The Assignment statement creates the variable Mileage. Use conditional processing to create a list of BY groups that contains types of cars where Mean values are greater than 20. The variable Mileage contains the results. Bygrps is the variable that holds the results of the Summary action.

2 The DESCRIBE statement prints the structure of the variable Mileage to the SAS log.

3 The PRINT statement prints the values of the variable Mileage. Because Mileage is an array, it is printed to the SAS log.

Output 6.5 *Data Structure of the Mileage Variable*

```
array ( 3 entries, 3 used);
 [ 1] string;
 [ 2] string;
 [ 3] string;
```

Output 6.6 *Value of Mileage Variable*

```
{Type=Hybrid,Type=Sedan,Type=Wagon}
```

See Also

- "DESCRIBE Statement" in *SAS Cloud Analytic Services: CASL Reference*
- "PRINT Statement" in *SAS Cloud Analytic Services: CASL Reference*

7

CASL Arrays

Overview

CASL arrays have the following qualities:

- An array is an ordered list of data.
- An array is accessed by position.
- A field in an array can store simple data types such as DOUBLE, INT64, and so on.
- A field in an array can store another CASL array. This effects a second dimension to the data structure. Multiple dimensions are supported.

- A field in an array can store other complex data types such as a dictionary or a result table.
- An array can store heterogeneous data types. However, most use cases store a single data type.

Operators

An array has any number of fields. Each field in an array stores a value. The fields are accessed by position. The fields begin numbering at 1.

Operator	Definition	Syntax
{*array-members*}	The { and } characters are used to begin and end an array definition.	*array-name* = { *value-1* <, *value-2*, …>}; **TIP** The comma as a field separator is optional.
${*literal-mode*}	The ${ operator is used to begin an array using literal mode. The values are accepted as-is, CASL does not attempt to evaluate *char-value* as a variable name. The } operator ends the array definition.	*array-name* = ${*char-value-1* <, *char-value-2* , …>};
+ ||	Appends one array to the end of another array.	*new-array* = *array-1* + *array-2*;
– !{*field-positions*}	Excludes the members of one array from another array. **Note:** The – operator can be used as the character range operator on page 46.	*new-array* = *array-1* – *array-2*; *new-array* = *array-1*[!{*field-positions*}];
/	Combines two arrays and returns a new array with the unique values.	*new-array* = *array-1* / *array-2*;
&	Selects the values that two arrays have in common and returns a new array.	*common-values* = *array-1* & *array-2*;
:	The range operator can be used to create a numeric array of 64-bit integers.	*new-array* = *lower-bound*:*upper-bound*;

Operator	Definition	Syntax
		Note: Do not enclose the range in braces.
== !=	The equality operator compares two arrays and returns TRUE when both arrays contain the same values in the same positions. Otherwise, FALSE is returned. When an array is compared with a single value and the value occurs in the array, an array of indexes of the value are returned. If the value does not occur, then nothing is returned. See "Check Whether an Array Contains a Value" on page 50.	*boolean = array-1 == array-2* *field-positions = array == value*
array-name[*field-positions*]	Subsets an array by selecting fields. This can also be used with exclusion and for assignment.	*subset-array = array*[*lower-bound*:*upper-bound*]; *subset-array = array*[{*field-positions*}]

Basic Syntax

The basic syntax for creating an array is the following:

array-name = {*value-1* <, *value-2* …>};
array-name[*position*] = *value*;

Note: The comma that is shown to separate values is optional.

Numeric Arrays

Example Code 7.1 *Assign Numeric Values by Position*

```
a[1] = 1;
a[2] = 2;
```

```
a[3] = 1 + 2;
```

The array positions are not required to be contiguous. However, the practical uses for arrays are either extraction of values from the results of an action or supplying an array of values to an action. As a result, contiguous positions are shown in most examples.

Example Code 7.2 *Assign Numeric Values with Braces*

```
b = {1, 2, 3};
```

Example Code 7.3 *Assign Numeric Values with the Range Operator*

```
five = 1:5;
print five;
```

The PRINT statement shows the following in the SAS log:

```
{1,2,3,4,5}
```

Character Arrays

Common Use

Specifying column names is a common use for character arrays.

Example Code 7.4 *Assign Character Values with Braces*

```
colNames = {"make", "model", "drivetrain"};
```

Like numeric arrays, you can assign values by position:

Example Code 7.5 *Assign Character Values by Position*

```
colNames[1] = "make";
colNames[2] = "mode";
colNames[3] = "drivetrain";
```

Variable Mode

As shown in the preceding section, you can specify a series of strings, enclosed in quotation marks. When a string is enclosed in quotation marks, CASL does not attempt to evaluate the string as a CASL variable name.

By default, CASL uses variable mode for arrays. The next example shows the following:

- Any string that is not enclosed in quotation marks is treated as a CASL variable.

- Because X is not enclosed in quotation marks, CASL treats it as a variable name. In this case, the variable is evaluated and colNames becomes an array with three string values: Make, Model, and Drivetrain.

Example Code 7.6 *Variable-Mode Array*

```
x = "model";
colNames = {"make", x, "drivetrain"};
```

Literal Mode

Literal mode provides a convenient shortcut if you need to specify many strings and want to avoid enclosing each string in quotation marks. The next example shows the following:

- The ${ } operator is used to enter literal mode.
- In literal mode, strings are used as-is. CASL does not attempt to evaluate the strings Make, Model, and Drivetrain as CASL variable names. When using literal mode, it is as if the strings are enclosed in quotation marks.

Example Code 7.7 *Literal-Mode Array*

```
colNames = ${make, model, drivetrain};
```

Combining Literal Mode and Variable Mode

If you need to combine variable mode and literal mode, you can escape from literal mode with the $(*variable-name*) operator. The next example shows the following:

- The ${ } operator is used to enter literal mode. The strings Make and Drivetrain are used as-is. CASL does not attempt to evaluate them as CASL variable names.
- The $( ) operator is used to escape literal mode. The variable X is evaluated and the string Model is substituted in its position in the array.

Example Code 7.8 *Escaping Literal Mode*

```
x = "model";
colNames = ${make, $(x), drivetrain};
```

Beware of accidentally nesting arrays when escaping literal mode. In cases like the following, appending arrays provides the expected outcome.

```
x = "model";                                    /* 1 */
drive_origin = {"drivetrain", "origin"};        /* 2 */
bad = ${make, $(x), $(drive_origin)};           /* 3 */
mmdo = ${make, $(x)} + drive_origin;            /* 4 */

print "Bad:  " bad;
print "Mmdo: " mmdo;
```

1 The variable X is assigned the string value Model.

2 Drive_origin is a character array with two values.

3 The Bad array shows an example of accidental nesting. The following statement avoids accidental nesting.

4 The Mmdo array is created by appending two arrays. Instead of including Drive_Origin inside the first set of braces, the + operator is used to append the array to the first array.

In the example below, the highlighted braces show the accidental nesting. The PRINT statement shows the following in the SAS log:

```
Bad:  {make,model,{drivetrain,origin}}
Mmdo: {make,model,drivetrain,origin}
```

Character Array Ranges

In some cases, columns are renamed to more uniform names. Within literal mode, you can specify a prefix and use the range operator, –, to create a range of character strings.

The rules for creating a character array with a range of values are as follows:

- Prefixes must match.
- Prefixes must end in a number.
- The first number must be less than the second number.

Example Code 7.9 *Specify a Range of Strings*

```
intervalCols = ${x1-x10 name5-name10};
print intervalCols;
```

The PRINT statement shows the following in the SAS log:

```
{x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,name5,name6,name7,name8,name9,name10}
```

Examples

Determine the Array Length

The DIM function is used to determine the length of an array.

```
ten = 1:10;                              /* 1 */
print dim(ten);                          /* 2 */
```

1 The range operator (:) is used to assign the values 1 to 10 to an array that is named Ten.

2 The DIM function returns the length of the array.

The PRINT statement shows the following in the SAS log:

```
10
```

Append Arrays

Example Code 7.10 *Append Numeric Arrays*

```
first_five = 1:5;                          /* 1 */
one_to_seven = first_five + {6, 7};        /* 2 */
print one_to_seven;
```

1 The range operator (:) is used to assign the values 1 to 5 to an array that is named First_five.

2 The concatenation operator (+) is used to append a second array to First_five. The new array is assigned to the variable that is named One_to_seven.

The PRINT statement shows the following in the SAS log:

```
{1,2,3,4,5,6,7}
```

Example Code 7.11 *Append a Single Value to an Array*

```
one_to_eight = one_to_seven + 8;           /* 1 */
print one_to_eight;
```

1 When a single value (8) is appended to an array, a new array is created. The new array combines the original array and the additional value.

The PRINT statement shows the following in the SAS log:

```
{1,2,3,4,5,6,7,8}
```

CASL also enables you to append character arrays. The + and || operators are used to append arrays.

Example Code 7.12 *Append Character Arrays*

```
make_model = {"make", "model"};            /* 1 */
drive_origin = ${drivetrain, origin};      /* 2 */

four_columns = make_model + drive_origin;  /* 3 */

print four_columns;
```

1 An array of two string values is assigned to the variable Make_model. The { } operators are used to specify variable mode. As a result, string values must be enclosed in quotation marks to prevent CASL from evaluating them as CASL variable names.

2 Another array of two string values is assigned to the variable Drive_origin. The ${ } operators are used to specify literal mode. In literal mode, you do not need to enclose strings in quotation marks.

3 The append operator (+) is used to create a new array that appends the values of the Drive_origin array to the values of the Make_model array. The new array is assigned to the CASL variable that is named Four_columns.

The PRINT statement shows the following in the SAS log:

```
{make,model,drivetrain,origin}
```

TIP Those are four column names from the Cars table and can be used with the fetchVars parameter: `table.fetch / table="cars" fetchVars=four_columns;`.

Subset Arrays

You can subset the values in one array with the values from another array:

Example Code 7.13 *Subset Numeric Arrays*

```
first_five = 1:5;                          /* 1 */
odds = {1, 3, 5};                          /* 2 */

evens = first_five - odds;                 /* 3 */
print evens;
```

1 The range operator (:) is used to assign the values 1 to 5 to an array that is named First_five.

2 An array that is named Odds is defined with the odd numbers 1, 3, and 5.

3 The subset operator (–) is used to create a new array. The new array is the result of removing the values of the Odds array that are present in the First_five array.

The PRINT statement shows the following in the SAS log:

```
{2,4}
```

Example Code 7.14 *Subset Character Arrays*

```
mmdo = ${make, model, drivetrain, origin};  /* 1 */
three_columns = mmdo - {"origin"};          /* 2 */
print three_columns;
```

1 The Mmdo array is created using the literal-mode operator, ${ }. The array contains four string values.

2 The subset operator (–) is used to create a new array. The new array is the result of removing one value, Origin, from the values that are present in the Mmdo array.

> **TIP** The second argument is an array with a single value. In this case, the braces are optional.

The PRINT statement shows the following in the SAS log:

```
{make,model,drivetrain}
```

Example Code 7.15 *Subset Arrays by Position*

```
mmdo = ${make, model, drivetrain, origin};  /* 1 */
model_drivetrain = mmdo[{2,3}];             /* 2 */
print model_drivetrain;
```

1 The Mmdo array is created using the literal-mode operator, ${ }. The array contains four string values.

2 The bracket operators ([ ]) are used to access the Mmdo array by position. A numeric array ({2,3}) is the expression that specifies the positions to access. A new array with the two values is returned and assigned to Model_drivetrain.

The PRINT statement shows the following in the SAS log:

```
{model,drivetrain}
```

Merge Arrays and Return Unique Values

```
four = 1:4;                                 /* 1 */
more = {6, 3, 5, 5};                        /* 2 */

uniq = four / more;                        /* 3 */
print uniq;
```

1 The range operator (:) is used to assign the values 1 to 4 to an array that is named Four.

2 The braces operators are used to assign four values to the array that is named More. The value that is in the second position, 3, is common with the values in the first array. The values in positions 3 and 4 are duplicates.

3 The / operator is used to create a new array from the values of Four and More. The new array contains the unique values. The new array is assigned to the variable that is named Uniq.

The PRINT statement shows the following in the SAS log:

```
{1,2,3,4,6,5}
```

Check Whether an Array Contains a Value

When the equality operator is used to compare a single value with an array, CASL returns the positions of the value in the array. You can use this behavior to determine whether an array contains a value.

Example Code 7.16 *Check for a Value in an Array*

```
a = {2, 2, 5, 2, 9};

if (a == 2) then do;                          /* 1 */
  print "The array contains 2.";
  print "The value occurs at the following positions:";
  do pos over (a == 2);                       /* 2 */
    print pos;
  end;
end;

if (a != 3) then print "The array does not contain 3.";
run;
```

1 The equality operator returns the positions of the number in the array. The IF statement evaluates any value as TRUE and executes the statements inside the IF block.

2 A DO OVER loop iterates over the positions of the number in the array and assigns the position to the variable Pos. The position is then printed.

```
The array contains 2.
The value occurs at the following positions:
1
2
4
The array does not contain 3.
```

The equality operator also applies to arrays with character values.

Example Code 7.17 *Check for a Value in a Character Array*

```
caslibNames = {"Formats", "ModelPerformanceData", "Models", "Public",
"Samples"};

if (caslibNames == "Public") then do;
  print "The Public caslib is in the list.";
  print "The value occurs at the following position:";
  do pos over (caslibNames == "Public");
    print pos;
  end;
end;

if (caslibNames != "Custom") then print "The array does not contain
Custom.";
run;
```

Common caslib names are stored in an array. To get the caslib names from a server and store them in an array, see the next example.

```
The Public caslib is in the list.
The value occurs at the following positions:
4
The array does not contain Custom.
```

Extract an Array from a Result Table

You can extract values from a single column of a result table as an array.

Example Code 7.18 *Assign Values from a Result Table*

```
table.caslibInfo result=r;                  /* 1 */

caslibNames = r.caslibinfo['Name'];         /* 2 */

print caslibNames;

do name over caslibNames;                   /* 3 */
  table.tableInfo / caslib=name;            /* 4 */
end;
run;
```

1 The table.caslibInfo action is run. The action identifies the caslibs that are available to the session and stores the results in the variable that is named R.

2 The data type for the R variable is a dictionary. The result table that is returned from the table.caslibInfo action is accessed from the dictionary with the dot operator. When the bracket operator is used with a result table to select a single column, the resulting data type is an array. In this instance, it is an array of strings that identify the caslib names.

3 A DO OVER loop iterates over the array of strings. During each iteration, the string value—the caslib name—is assigned to a temporary variable that is named Name.

4 For each iteration of the DO OVER loop, the table.tableInfo action is run to list all the in-memory tables in the specified caslib. The caslib is identified during each iteration by the value of the Name variable.

Output 7.1 *Values of the caslibNames Variable*

```
{CASUSER(sasdemo),Formats,Models,Public,Samples}
```

Note: The results for each table.tableInfo action are not shown.

Creating Multidimensional Arrays

The following example creates a two- and three-dimensional array using the CAS language.

```
proc cas;

/* a 2-dimensional, 3x3 array */
x={ {1,2,3}, {4,5,6}, {7,8,9} };

/* these are the same element */
print x[1][1];
print x[1,1];

/* these are the same element */
print x[3][3];
print x[3,3];
run;

/* a 3-dimensional, 3x3x3 array */
x={ { {01,02,03}, {04,05,06}, {07,08,09} },
    { {10,11,12}, {13,14,15}, {16,17,18} },
    { {19,20,21}, {22,23,24}, {25,26,27} } };

/* these are all the same element */
print x[1,1,1];
print x[1][1][1];
print x[1,1][1];
print x[1][1,1];
run;

/* these are all the same element */
print x[3,3,3];
print x[3][3][3];
print x[3,3][3];
print x[3][3,3];
run;


quit;
```

8

CASL Dictionaries

Overview

CASL dictionaries have the following qualities:

- A dictionary is an unordered list of key-value pairs.
- A dictionary is accessed by key. Dictionary keys are strings. The keys are case-insensitive.
- Dictionary values can be any data type, including a result table, array, or another dictionary.

Most actions use a dictionary to return results from the server to the client. To work with the results of an action, it is important to understand CASL dictionaries.

Basic Syntax

Dictionaries use either brackets ([and]) or the dot operator (.). The basic syntax for creating a dictionary is the following:

dictionary-name["*key*"] = *value*;

dictionary-name.key = *value*;

The next code sample accomplishes the following:

- A variable that is named HmeqTable becomes a dictionary with two keys.
- One key is Name, and the other key is Copies.
- The value that is paired with Name is a variable of data type STRING with the value Hmeq.
- The value that is paired with the Copies is a variable of data type INT64 with a value of 2.

Example Code 8.1 *Using Brackets to Specify Dictionary Keys*

```
hmeqTable["name"] = "hmeq";
hmeqTable["copies"] = 2;
```

> **TIP** When assigning or retrieving values with the bracket operators, enclose the key in quotation marks so that CASL does not attempt to interpret the key as a CASL variable name. As an alternative, you can specify a CASL variable that has the string data type.

The following code sample has the same outcome as the preceding sample. The difference is that the dot operator is used to specify the dictionary keys.

Example Code 8.2 *Using the Dot Operator to Specify Dictionary Keys*

```
hmeqTable.name = "hmeq";
hmeqTable.copies = 2;
```

Examples

View the Data Structure of a Dictionary

The data structure of the HmeqTable dictionary can be viewed with the DESCRIBE statement:

```
   hmeqTable.name = "hmeq";
   hmeqTable.copies = 2;

   describe hmeqTable;
run;
```

The PRINT statement shows the following in the SAS log:

```
dictionary ( 2 entries, 2 used);
 [name] string;
 [copies] int64_t;
```

For more examples, see “View the Data Structure and Values of a Variable” on page 38.

Print Keys and Values

To view the keys and values, you can use the PRINT statement alone:

```
    hmeqTable.name = "hmeq";
    hmeqTable.copies = 2;

    print hmeqTable;
run;
```

The PRINT statement shows the following in the SAS log:

```
{name=hmeq,copies=2}
```

Note: The information can become difficult to read under two conditions: the number of keys increases or the values are arrays or other dictionaries.

Iterate Over Key-Value Pairs in a Dictionary

You can also use the DO OVER statement to iterate over the keys and values:

```
    do k,v over hmeqTable;
        print "Key: " put(k, $10.) " Value: " v;
    end;
run;
```

The PRINT statement shows the following in the SAS log:

```
Key: name          Value: hmeq
Key: copies        Value: 2
```

Access One Value from a Dictionary

You can use the brackets or dot operator to access a value:

```
table_name = hmeqTable["name"];

table_name = hmeqTable.name;
```

Delete a Dictionary Key

You can use the DELETE statement to remove the value that is associated with a key:

```
   hmeqTable.name = "hmeq";          /* 1 */
   hmeqTable.caslib = "casuser";     /* 2 */
   print hmeqTable;
run;

   delete hmeqTable.caslib;          /* 3 */
   print hmeqTable;
run;
```

1 A dictionary key, Name, is paired with the value Hmeq.

2 A second key is added to the dictionary. The key is named Caslib and is paired with the value Casuser.

3 The DELETE statement removes the value that is associated with the Caslib key.

The PRINT statement shows the following in the SAS log:

```
{name=hmeq,caslib=casuser}
{name=hmeq}
```

9

Loading Data with CASL

Overview

You can use PROC CAS to access and load your data locally or remotely, to the CAS server. Loading data from a client-side file with the UPLOAD statement is appropriate for small data sets and when you are learning to program with actions. When the data sets become larger, it is more efficient to use a server-side load with the loadTable action to access data. To load data, you must associate a caslib with a data source.

Once you have loaded your data into CAS as an in-memory table, you can save the table to a data source as a file.

Terminology

The following terms are used throughout this section:

client-side load
: data that is transferred from SAS to the CAS server.

data source
: a table, view, directory, or file from which information is extracted. For example, the data source can be a directory or the host, port, and other connection information for an Oracle database.

server-side load
: data that is read from a caslib's data source by the CAS server. Common data sources are file system directories that are accessible to the CAS controller. Other data sources are accessed through SAS Data Connectors.

caslib
: an in-memory space to hold tables, access control lists, and data source information. All data is available to CAS through caslibs and all operations in CAS that use data are performed with a caslib in place.

Casuser
: a personal caslib that is always available and has global scope. When you start a CAS session without specifying a caslib, Casuser becomes the active caslib for that session. The host CAS user location is `~/casuser/` where the `~` represents the user's home directory. When data is loaded in the Casuser caslib, only sessions started by you can access the in-memory tables.

in-memory table
: data that has been loaded into memory. CAS actions operate on in-memory tables only.

Examples

Load a Client-Side File

The following example uses the HTTP procedure to download the Cars data set locally into a temporary file. It then uses the PROC CAS UPLOAD statement to load the file into CAS. The file received by CAS is stored as temporary until the file is loaded to an in-memory table. Once it is loaded, the file is removed.

This example uses the caslib Casuser. Casuser is one of the personal caslibs that is available when a CAS session is started.

```
filename cars
temp;                                                    /* 1 */
proc
http                                                              /
* 2 */
   url='http://support.sas.com/documentation/onlinedoc/viya/
exampledatasets/cars.csv'
   out=cars;
run;

proc cas;
```

```
upload
 /* 3 */
      path=%sysfunc(quote(%sysfunc(pathname(cars))))
      casOut={caslib="casuser", name="cars"}
      importOptions="csv";
run;


table.tableInfo /
 /* 4 */
      caslib="casuser",
      table="cars";
quit;

filename cars
clear;                                                    /* 5 */
```

1 The FILENAME statement creates the fileref Cars to the SAS temporary folder for the cars.csv file.

2 The HTTP procedure creates a GET request for the cars.csv file. For information about the HTTP procedure, see "HTTP Procedure" in *Base SAS Procedures Guide*.

3 The UPLOAD statement uploads the file cars.csv to CAS as the in-memory table Cars. The PATH= option uses the %SYSFUNC function to specify the path to the input file. The CASOUT= option specifies that the in-memory table name is Cars, and it is stored in the Casuser caslib. The IMPORTOPTIONS= option specifies the file type of the input file. For information about the UPLOAD statement, see "UPLOAD Statement" in *SAS Cloud Analytic Services: CASL Reference*.

4 The tableInfo action displays information about the in-memory table Cars. For information about the tableInfo action, see "Table information" in *SAS Viya: System Programming Guide*.

5 The FILENAME statement clears the Cars fileref.

Output 9.1 *SAS Log*

```
NOTE: Cloud Analytic Services made the uploaded file available as table CARS in
caslib CASUSER(sasdemo).
NOTE: The table CARS has been created in caslib CASUSER(sasdemo) from binary data
uploaded to Cloud Analytic Services.
{caslib=CASUSER(sasdemo),tableName=CARS}
```

Output 9.2 *Results of the table.tableInfo Action*

Table Information for Caslib CASUSER(sasdemo)										
Table Name	Number of Rows	Number of Columns	Number of Indexed Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Compressed
CARS	428	15	0	utf-8	2019-03-15T13:20:51-04:00	2019-03-15T13:20:51-04:00	No	No	No	No

Add a Caslib and Load a Server-Side File

The following example uses the loadTable action to perform a server-side load of the file hmeq.csv, and then displays information about the in-memory table that is created. The file is loaded into the caslib Sampledata as the in-memory table Hmeq, For more information, see "Working with Tables and Files" in *SAS Viya: System Programming Guide*.

First, download the hmeq.csv file from SAS Viya Example Data Sets and save it in a data source that CAS can access.

```
proc cas;
   table.addCaslib /                                    /*1*/
      name="sampledata"
      dataSource={srcType="path"}
      path="file-system-directory-path";

   table.loadTable /                                    /*2*/
      caslib="sampledata"
      path="hmeq.csv"
      casout={name="hmeq"};
run;

   table.columninfo / table="hmeq";                     /*3*/
   table.tableDetails / table="hmeq";                   /*4*/
   table.dropCaslib / caslib="sampledata";              /*5*/
quit;
```

1 The table.addCaslib action adds a caslib that has access to your data. The input data must be located in a directory that is accessible to the CAS server. For information about the addCaslib action, see "Add caslib" in *SAS Viya: System Programming Guide*.

2 The table.loadTable action loads the input data set hmeq.csv into CAS. An in-memory table named Hmeq is created in the caslib Sampledata. For information about the loadTable action, see "Load table" in *SAS Viya: System Programming Guide*.

3 The table.columInfo action displays column information such as type, length, and width. For information about the columnInfo action, see "Column information" in *SAS Viya: System Programming Guide*.

4 The table.tableDetails action displays details of table Hmeq. For information about the tableDetails action, see "Table details" in *SAS Viya: System Programming Guide*.

5 The table.dropCaslib action drops the caslib Sampledata.

Output 9.3 *SAS Log*

```
NOTE: Active Session now casauto.
NOTE: 'sampledata' is now the active caslib.
NOTE: Cloud Analytic Services added the caslib 'sampledata'.
 NOTE: Cloud Analytic Services made the file hmeq.csv available as table HMEQ in
caslib sampledata.
 {caslib=sampledata,tableName=HMEQ}
```

Output 9.4 *Results of the table.addCaslib Action*

CAS Library Information					
Library	Type	Path	Sub-directories included	Session local	Active
sampledata	PATH	/rdstore/data/sampledata/	No	Yes	Yes

Output 9.5 *Results of the table.columnInfo Action*

Column Information for HMEQ in Caslib sampledata						
Column	Id	Type	Length	Formatted Length	Format Width	Format Decimal
BAD	1	double	8	12	0	0
LOAN	2	double	8	12	0	0
MORTDUE	3	double	8	12	0	0
VALUE	4	double	8	12	0	0
REASON	5	varchar	7	7	0	0
JOB	6	varchar	7	7	0	0
YOJ	7	double	8	12	0	0
DEROG	8	double	8	12	0	0
DELINQ	9	double	8	12	0	0
CLAGE	10	double	8	12	0	0
NINQ	11	double	8	12	0	0
CLNO	12	double	8	12	0	0
DEBTINC	13	double	8	12	0	0

Output 9.6 *Results of the table.tableDetails Action*

Detail Information for hmeq in Caslib sampledata.														
Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated	Index Size	Compressed Size	Compression Ratio
ALL	280	140	5960	785334	70134	140	810424	140	810424	0	0	0	0	0

Save and Drop an In-Memory Table

The following example loads the Iris data set into CAS, and then saves it to a data source. The active caslib is Casuser. The file is uploaded to Casuser as an in-memory table, and then saved in Casuser as a SASHDAT file. Once the table is saved as a SASHDAT file, you can drop the in-memory table. For more information about saving tables, see "Saving Tables" in *SAS Viya: System Programming Guide*.

```
filename iris
temp;                                                    /* 1 */
proc
http                                                              /
* 2 */
   url='http://support.sas.com/documentation/onlinedoc/viya/
exampledatasets/iris.csv'
   out=iris;
```

```
run;

proc cas;
   upload path=
%sysfunc(quote(%sysfunc(pathname(iris))))                    /* 3 */
      casOut={caslib="casuser", name="iris"}
      importOptions="csv";
   run;


table.save /
 /* 4 */
      table={caslib="casuser", name="iris"}
      name="iris.sashdat";
   run;

table.fileInfo /
 /* 5 */
      caslib="casuser"
      path="iris.sashdat";
   run;


table.dropTable /
 /* 6 */
      caslib="casuser"
      name="iris";
run;
filename iris
clear;                                                        /* 7 */
```

1 The FILENAME statement creates the fileref Iris to the SAS temporary folder for the iris.csv file.

2 The HTTP procedure creates a GET request for the iris.csv file.

3 The UPLOAD statement uploads the file iris.csv to CAS as the in-memory table Iris. The PATH= option uses the %SYSFUNC function to specify the path to the input file. The CASOUT= option specifies that the in-memory table name is Iris, and it is stored in the Casuser caslib. The IMPORTOPTIONS= option specifies the file type of the input file.

4 The save action saves the table Iris to the data source as the file iris.sashdat. The table parameter specifies the caslib and name of the in-memory table. The name parameter specifies the new file name. For information about the save action, see “Save table” in *SAS Viya: System Programming Guide*.

5 The fileInfo action displays information about the file iris.sashdat.

6 The dropTable action drops the table Iris that is in the caslib Casuser. For information about the dropTable action, see “Drop table” in *SAS Viya: System Programming Guide*.

7 The FILENAME statement clears the Iris fileref.

Output 9.7 *SAS Log: Table Saved*

```
 NOTE: Cloud Analytic Services saved the file iris.sashdat in caslib
CASUSER(sasdemo).
 {caslib=CASUSER(sasdemo),name=iris.sashdat}
```

Output 9.8 *Results of the table.fileInfo Action*

File Information for iris.sashdat in caslib sampledata.					
Permission	Owner	Group	Name	Size of File in Bytes	Time
-rwxr-xr-x	sasdemo	users	iris.sashdat	413936	2019-03-15T11:41:56-04:00

Output 9.9 *SAS Log: Table Dropped*

```
NOTE: Active Session now CASAUTO.
NOTE: Cloud Analytic Services dropped table iris from caslib CASUSER(sasdemo).
```

Key Ideas

- CAS actions operate on in-memory tables only. In-memory tables are available by using caslibs.
- All data is available to CAS through caslibs and all operations in CAS that use data are performed with a caslib in place. The caslib that is active when you first start a session is Casuser.
- Loading data from a client-side file with the UPLOAD statement is appropriate for small data sets. For large data, using a server-side load with the loadTable action is more efficient.
- The Tables action set contains actions that enable you to load, save, drop, and manage your in-memory tables.

See Also

- “Server-Side Data Access” in *SAS Cloud Analytic Services: Fundamentals*
- “Client-Side Data Access” in *SAS Cloud Analytic Services: Fundamentals*
- “Caslibs” in *SAS Cloud Analytic Services: Fundamentals*
- “UPLOAD Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “Table Action Set” in *SAS Viya: System Programming Guide*
- “HTTP Procedure” in *Base SAS Procedures Guide*

- “Caslibs” in *SAS Cloud Analytic Services: Fundamentals*
- “ Syntax for Selected Functions Used with the %SYSFUNC and %QSYSFUNC Functions” in *SAS Macro Language: Reference*
- “Data Life Cycle” in *SAS Cloud Analytic Services: Fundamentals*

10

Running Actions with CASL

Overview

CAS actions are executable routines that the CAS server makes available to client programs. They are the smallest unit of work for the CAS server. There are actions for each of the many analytic algorithms, for data management, for administration, and for simple housekeeping. Actions are organized into groups called action sets. For example, the Table action set contains actions for loading tables, deleting tables, managing table attributes, and so on.

You can use actions to perform system tasks such as managing CAS sessions, loading tables, and performing administrative tasks. The majority of actions return

results. The results that an action returns can be one of the following types of information:

- result objects
- messages
- return status
- performance statistics

Once your results are returned, you can use CASL language functionality to extract and manipulate information from the results. You can then use that information as input to another action, or simply view the results.

Basic Syntax

Use the ACTION statement in PROC CAS to specify an action. The basic syntax for the ACTION statement is the following:

<ACTION> <*action-set-name.*>*action-name* <RESULT= <*variable*>
<STATUS = <*rc*> < / *parameters*>;

This form of the action execution enables you to specify where the results are placed. If a result is not specified, the results are printed to the default ODS destination. The status of the action is placed either in the status variable supplied or in the _status variable. For complete documentation for the ACTION statement, see "ACTION Statement" in *SAS Cloud Analytic Services: CASL Reference*. For a complete list of actions and action sets, see *SAS Viya Actions and Action Sets by Name and Product*.

Although a typical CASL program might use many actions, the following is a simple example of a CASL program that invokes just two actions.

Note: Before you can work with an in-memory table in CAS, you must download the table Hmeq.csv and load it into CAS. For the complete code for accessing the Hmeq data set, see "Add a Caslib and Load a Server-Side File" on page 60.

```
   table.columninfo / table="hmeq";          /*1*/
run;
   table.fetch / to=5 table="hmeq";          /*2*/
run;
quit;
```

1 The columnInfo action displays column information for table Hmeq.

2 The table.Fetch action displays the results.

Output 10.1 *Results: ColumnInfo Action for the Hmeq Table*

Results from table.columnInfo

Column Information for HMEQ in Caslib CASUSER(sasdemo)						
Column	Id	Type	Length	Formatted Length	Format Width	Format Decimal
BAD	1	double	8	12	0	0
LOAN	2	double	8	12	0	0
MORTDUE	3	double	8	12	0	0
VALUE	4	double	8	12	0	0
REASON	5	varchar	7	7	0	0
JOB	6	varchar	7	7	0	0
YOJ	7	double	8	12	0	0
DEROG	8	double	8	12	0	0
DELINQ	9	double	8	12	0	0
CLAGE	10	double	8	12	0	0
NINQ	11	double	8	12	0	0
CLNO	12	double	8	12	0	0
DEBTINC	13	double	8	12	0	0

Output 10.2 *Results: Fetch Action for the Hmeq Table*

Results from table.fetch

Selected Rows from Table HMEQ													
Index	BAD	LOAN	MORTDUE	VALUE	REASON	JOB	YOJ	DEROG	DELINQ	CLAGE	NINQ	CLNO	DEBTINC
1	1	1100	25860	39025	HomeImp	Other	10.5	0	0	94.366666667	1	9	.
2	1	1300	70053	68400	HomeImp	Other	7	0	2	121.83333333	0	14	.
3	1	1500	13500	16700	HomeImp	Other	4	0	0	149.46666667	1	10	.
4	1	1500	.	.			.	.	.	.	.	.	.
5	0	1700	97800	112000	HomeImp	Office	3	0	0	93.333333333	0	14	.

Action Parameters

Overview

Each action can accept zero or more parameters. Each parameter has a name and a data type. Data types on page 3 can be arrays, dictionaries, strings, or any combination. The following svmTrain action has a variety of data types as input values.

Example Code 10.1 *Specifying Parameter Values: Expanded Form*

```
action
svm.svmTrain /                                                    /
*1*/
   table={name='hmeq',
caslib='casuser',                                        /*2*/
      importOptions={fileType="csv", allowTruncation=false,
guessRows=50}
      },
   inputs={'loan', 'mortdue', 'value', 'yoj', 'clage', 'clno',
'debtinc',    /*3*/
           'reason', 'job', 'derog', 'delinq', 'ninq'},
   nominals={'bad', 'reason', 'job', 'derog', 'delinq',
'ninq'},             /*4*/

target='bad'
   /*5*/
;
run;
```

1. Svm.svmTrain specifies SVM as the action set name and svmTrain is the action.
2. The table parameter specifies information about the input table. The parameter value list `{name='hmeq', caslib='casuser', importOptions={fileType="csv", allowTruncation=false, guessRows=50}}` is a multidimensional dictionary of strings with a nested dictionary.
3. The inputs parameter specifies the variables to use for analysis. The parameter value `{'loan', ..., 'ninq'}` is an expanded form of specifying an array.
4. The nominals parameter specifies the variables to use as nominals. The parameter value `{'bad',...,'ninq'}` is an expanded form of specifying an array.
5. The target parameter specifies the target variable to use for analysis. The parameter value `'bad'` is a string.

Use Arrays to Condense Long Lists

You can also condense an array of parameter values by assigning the array to a variable. In the following example, the variable cols contains the input column names. The nomVals variable contains the nominal value names.

```
cols={'loan', 'mortdue', 'value', 'yoj', 'clage', 'clno',
              'debtinc', 'reason', 'job', 'derog', 'delinq', 'ninq'};
nomVals={'bad', 'reason', 'job', 'derog', 'delinq', 'ninq'};
```

Here is the example from the section above. However, in the following example, the variables are used as values to the input and nominals parameters.

```
action
svm.svmTrain /
   table={name='hmeq',
caslib='casuser',
      importOptions={fileType="csv", allowTruncation=false,
guessRows=50}
      },

inputs=cols,

nominals=nomVals,


target='bad'

;
run;
```

Use Dictionaries as Parameter Values

Frequently, parameter values are specified as dictionaries and contain nested dictionaries. You can assign dictionary values to a variable and then use the variable as input to parameters. For example, the svmTrain action has a parameter named table that specifies the nested parameters name, caslib, and importOptions.

```
action svm.svmTrain / table={name='hmeq',
       caslib='casuser',
       importOptions={fileType="csv", allowtruncation=false,
guessrows=50}
        }
```

The following Assignment statement creates a variable named hmeqTable that contains the values for the parameters name, caslib, and importOptions.

```
hmeqTable.name='hmeq.csv';
hmeqTable.caslib='casuser';
hmeqTable.importOptions={fileType="csv", allowTruncation=false,
guessRows=50};
```

The following statement then specifies the variable hmeqTable as the value of the table parameter.

```
action svm.svmTrain / table=hmeqTable;
```

Combine Arrays and Dictionaries as Parameter Values

Column names have been shown in simple arrays of strings. When a column is specified by name only, the default format is applied. To specify a format, you can combine the convenience of arrays with a dictionary.

```
cols = {"bad", "reason", {name="loan", format="dollar."}};
table.fetch / table=hmeqTable fetchVars=cols;
```

Cols is an array with three fields. The first two fields use the STRING data type. The third field is a dictionary with the keys Name and Format.

Parameter List Shortcuts

Many parameters are defined as parameter lists. One example of a parameter list is the table parameter, which specifies an in-memory table that is used for input to the action. This parameter supports coercion. You can recognize a parameter list when you see several parameters enclosed in a set of single braces, as in the following table= parameter:

```
table={
     caslib="string",
     computedOnDemand= TRUE | FALSE,
     ...
     name="table-name",
     ...
}
```

Parameter lists each have a designated parameter. If it is the only parameter that you want to specify, you can drop the name of the parameter, along with the enclosing braces. For the example, the name parameter is the only required parameter and it is common programming practice to specify the table name only.

For example, the following two code samples are equivalent:

Example Code 10.2 *Long Form*

```
proc cas;
   session casauto;
   simple.summary /
      table={name="iris"};
   run;
```

TIP Use the long form if you want to specify non-default values for other parameters in the table parameter.

Example Code 10.3 *Shortcut Form*

```
proc cas;
   session casauto;
   simple.summary /
      table="iris";
   run;
```

> **TIP** Use the shortcut form if you want to specify only the required parameter and use default values for the others.

The second example is functionally equivalent to the first example because the name parameter is the only required parameter. The server coerces the only value that is provided for the list, "iris," into the parameter that accepts coercion. The parameters that accept coercion are identified in the action reference documentation with short examples of the long form and shortcut form.

Examples

Create a List of Tables and Dynamically Load the Tables into Memory

Frequently, CASL programs create result tables. CASL provides functions that work specifically with results tables. The following example uses the built-in function FINDTABLE and the common function SCAN. The example does the following:

- lists all the files available in the active caslib and stores the results.
- searches the variable for result tables.
- loops through the list and loads each source file into memory.

```
proc cas;
   table.fileinfo result = fileresult / caslib = "casuser";      /*1*/
   print fileresult;                                             /*2*/
   filelist = findtable(fileresult);                             /*3*/
   do cvalue over filelist;                                      /*4*/
      table.loadtable /                                          /*5*/
         caslib = "casuser"
         path = cvalue.name
         casout = {name = scan(cvalue.name,1,".")};
   end;
quit;
```

1 The fileInfo action lists all of the files available in the caslib Casuser and stores the result table in the variable Fileresult.

2 The PRINT statement prints the result table. The result table shows that there are four tables in the Casuser caslib.

3 The built-in function FINDTABLE returns the table from the results and stores it in the variable Filelist. The FINDTABLE function assumes that the argument is an array and iterates over the array to find the first result table. A copy of that table is returned.

4 The DO OVER statement iterates over the variable Filelist and returns a list of the names of the source files.

5 The loadTable action loads the tables into memory. The variable Cvalue.name contains the list of table names, and is used as the input for the path parameter. The name parameter specifies the name to associate with the table. The in-memory table name is created from the source file name using the SCAN function to remove the file extension.

Output 10.3 *Results: File Information for the Casuser Caslib*

fileresult: Results from table.fileInfo

File Information for root of caslib CASUSER(sasdemo).					
Permission	Owner	Group	Name	Size of File in Bytes	Time
-rwx------	sasdemo	Domain Users	cars.csv	42177	2018-02-19T16:21:44-04:00
-rwx------	sasdemo	Domain Users	heart.csv	465447	2018-03-20T14:41:51-04:00
-rwx------	sasdemo	Domain Users	humid.csv	10532	2018-03-20T14:41:54-04:00
-rwx------	sasdemo	Domain Users	junkmail.csv	707912	2018-03-20T14:41:57-04:00

The result in the log shows that all the tables in the source path of the caslib were dynamically loaded into memory with the same name as the source file.

Output 10.4 *Listing of Files Loaded into CAS*

```
NOTE: Active Session now MYCAS.
cars.csv
NOTE: Cloud Analytic Services made the file cars.csv available as table CARS in
caslib CASUSER(sasdemo).
{caslib=CASUSER(sasdemo),tableName=CARS}
heart.csv
NOTE: Cloud Analytic Services made the file heart.csv available as table HEART in
caslib CASUSER(sasdemo).
{caslib=CASUSER(sasdemo),tableName=HEART}
humid.csv
NOTE: Cloud Analytic Services made the file humid.csv available as table HUMID in
caslib CASUSER(sasdemo).
{caslib=CASUSER(sasdemo),tableName=HUMID}
junkmail.csv
NOTE: Cloud Analytic Services made the file junkmail.csv available as table
JUNKMAIL in caslib CASUSER(sasdemo).
{caslib=CASUSER(sasdemo),tableName=JUNKMAIL}
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.07 seconds
      cpu time            0.04 seconds
```

Store Action Results in a Variable

You can save the results of an action to a variable and then perform operations on the results inside the variable. You can then use the variable as input to another action or a CASL statement. The following example stores the results of the svmTrain action in a variable. Specific tables are selected from the variable and printed.

Note: Before you can work with an in-memory table in CAS, you must download the table Hmeq.csv and load it into CAS. For the complete code for accessing the Hmeq data set, see "Add a Caslib and Load a Server-Side File" on page 60.

```
proc cas;
   action svm.svmTrain
result=svmResults /                          /* 1 */
      table={name='hmeq', caslib='casuser'},
      inputs = ${loan, mortdue, value, reason, job, yoj,
            derog, delinq , clage , ninq, clno , debtinc}
      nominals = ${bad, reason, job, derog, delinq, ninq}
      target="bad";
   run;
   describe
svmResults;                                          /* 2 */
   run;
   print
svmResults.modelInfo                                    /* 3 */
         svmResults.misclassification
         svmResults.fitStatistics;
   run;
quit;
```

1 The Assignment statement assigns the result tables created by the svmTrain action to a variable named svmResults by using the RESULT= option

2 The DESCRIBE statement writes the data structure and data type of the svmResults dictionary to the SAS log.

3 The PRINT statement selects the tables to be printed to the output destination.

The DESCRIBE statement writes the structure of the svmResults dictionary into the SAS log. This output shows that there are six results tables and includes the row and column information for each. You can use this information to select specific tables. In this example, the ModelInfo, Misclassification, and FitStatistics tables are selected for printing.

Output 10.5 *Structure of the SvmResults Dictionary*

```
93            describe svmResults;
 94         run;
 dictionary ( 6 entries, 6 used);
  [ModelInfo] Table ( [8] Rows [2] columns
 Column Names:
  [1] Descr            [                ] (char)
  [2] Value            [                ] (char)
  [NObs] Table ( [2] Rows [2] columns
 Column Names:
  [1] Descr            [                ] (char)
  [2] N                [                ] (double)
  [TrainingResult] Table ( [10] Rows [2] columns
 Column Names:
  [1] Descr            [                ] (char)
  [2] Value            [                ] (double)
  [IterHistory] Table ( [22] Rows [3] columns
 Column Names:
  [1] Iteration        [                ] (double)
  [2] Complementarity  [                ] (double)
  [3] Feasibility      [                ] (double)
  [Misclassification] Table ( [3] Rows [4] columns
 Column Names:
  [1] Observed         [                ] (char)
  [2] PredEvent        [0               ] (double)
  [3] PredNonEvent     [1               ] (double)
  [4] TotalTrain       [Total           ] (double)
  [FitStatistics] Table ( [4] Rows [2] columns
 Column Names:
  [1] Statistic        [                ] (char)
  [2] Training         [                ] (double)
```

Output 10.6 *Results: SvmTrain Result Tables Printed Using the PRINT Statement*

modelInfo: Results from svm.svmTrain

Model Information	
Task Type	C_CLAS
Optimization Technique	Interior Point
Scale	YES
Kernel Function	Linear
Penalty Method	C
Penalty Parameter	1
Maximum Iterations	25
Tolerance	1e-06

Misclassification Matrix			
	Training Prediction		
Observed	0	1	Total
0	3055	9	3064
1	257	43	300
Total	3312	52	3364

Fit Statistics	
Statistic	Training
Accuracy	0.9209
Error	0.0791
Sensitivity	0.9971
Specificity	0.1433

TIP You can use the DEFAULT statement to specify a default variable that will store action's results. For more information, see "DEFAULT Statement" in *SAS Cloud Analytic Services: CASL Reference*.

Work with the Results of an Action

The following example uses three steps to perform the following tasks:

- Evaluate the data in the Hmeq table and store the results in the CASL variable cardResult.
- CASL statements are used to iterate over the CASL variable cardResult, and create two variables that contain arrays: one for the analysis variables, and one for the nominal variables. These CASL variables are then used as input to the inputs and nominal parameters in the svmTrain action.
- The svmTrain action from the Support Vector Machine (svm) action set builds a model for predicting home loan defaults. The action has three parameters that require input variables. The inputs parameter specifies the variables to be used

for analysis. The nominals parameter specifics nominal variables to use for analysis. The target parameter specifies the target variable to use for analysis.

Note: Before you can upload a table to CAS, you must download the table Hmeq.csv and load it into CAS. For the complete code for accessing the Hmeq data set, see "Add a Caslib and Load a Server-Side File" on page 60.

1 First, run the summarize action to create a cardinality data table, which contains summary information for each variable and some additional statistics about numeric variables. Use the fetch action to fetch the rows from the cardinality table and store them in a variable. The variable will be used for conditional processing of row and columns selection in the next step.

```
proc cas;
   cardinality.summarize /                                    /*1*/
      table={name='hmeq'},
      cardinality={name='Card' replace=true},
      options={nLevels=20};

   table.fetch result=cardResult / table="Card" to=99999;     /*2*/

   describe
cardResult ;                                              /*3*/
   run;
   print cardResult ;                                         /*4*/
   run;
```

1 The summarize action evaluates the data cardinality and creates an in-memory table named Card.

2 The fetch action displays the rows in the table Card. The RESULT= option specifies that the result table is stored the variable cardResult.

3 The DESCRIBE statement writes the data types and structure of the variable cardResult to the SAS log.

4 The PRINT statement displays the contents of the results table that is stored in the variable cardResult. Because the results are result tables, they are printed to the ODS destination.

The DESCRIBE statement prints the structure of the variable to the SAS log. The variable cardResult is a dictionary that contains the table named Fetch. Table Fetch contains the rows and columns from the Hmeq table.

Output 10.7 *Structure of the CardResult Variable*

```
dictionary ( 1 entries, 1 used);
 [Fetch] Table ( [13] Rows [27] columns
Column Names:
 [1] _Index_           [                ] (int32)
 [2] _VARNAME_         [Variable name   ] (char)
 [3] _FMTWIDTH_        [Width of the variable formatted value] (double)
 [4] _TYPE_            [Type of the raw values] (char)
 [5] _RLEVEL_          [Recommended level for analytics] (char)
 [6] _ORDER_           [Variable sort order] (char)
 [7] _MORE_            [Have more unreported levels] (char)
 [8] _CARDINALITY_     [Number of levels] (double)
 [9] _NOBS_            [Number of observations] (double)
 [10] _SUMFREQS_        [Total summation of frequencies] (double)
 [11] _NMISS_           [Number of missing values] (double)
 [12] _MISSFMT_         [Format of the missing value] (char)
 [13] _VISIBLE_         [Percentage of the visible part of the report] (double)
 [14] _MIN_             [Minimum numeric value] (double)
 [15] _MAX_             [Maximum numeric value] (double)
 [16] _MEAN_            [Mean            ] (double)
 [17] _STDDEV_          [Standard deviation] (double)
 [18] _SKEWNESS_        [Skewness        ] (double)
 [19] _KURTOSIS_        [Kurtosis        ] (double)
 [20] _MFREQ_           [Maximum frequency] (double)
 [21] _MFREQFOUNDLEVEL_ [Found maximum frequency in the visible part of the
report] (char)
 [22] _MFREQNUM_        [Numeric level with the maximum frequency] (double)
 [23] _MFREQCHR_        [Character level with the maximum frequency] (char)
 [24] _MFREQCFMT_       [Formatted level with the maximum frequency] (char)
 [25] _LASTNUM_         [Last raw numeric value (cutoff)] (double)
 [26] _LASTCHR_         [Last raw character value (cutoff)] (char)
 [27] _LASTCFMT_        [Last formatted value (cutoff)] (char)
1007  describe card;
```

The following partial results table shows the rows that were fetched from the Hmeq table and stored as the fetch action results table. In the next step, the columns Variable name (_VARNAME_) and Recommended level for analytics (_RLEVEL_) are used for selecting rows and columns to build an array of nominal variable names and an array of analysis variable names.

Output 10.8 *Partial Results: Fetched Rows from the Hmeq Table*

cardResult: Results from table.fetch

Index	Variable name	Width of the variable formatted value	Type of the raw values	Recommended level for analytics	Variable sort order	Hav unre leve	rma el v kin ue
1	BAD	12	N	CLASS	ASC	N	
2	LOAN	12	N	INTERVAL	ASC	Y	
3	MORTDUE	12	N	INTERVAL	ASC	Y	
4	VALUE	12	N	INTERVAL	ASC	Y	
5	REASON	7	C	CLASS	ASC	N	
6	JOB	7	C	CLASS	ASC	N	
7	YOJ	12	N	INTERVAL	ASC	Y	
8	DEROG	12	N	CLASS	ASC	N	
9	DELINQ	12	N	CLASS	ASC	N	
10	CLAGE	12	N	INTERVAL	ASC	Y	
11	NINQ	12	N	CLASS	ASC	N	
12	CLNO	12	N	INTERVAL	ASC	Y	
13	DEBTINC	12	N	INTERVAL	ASC	Y	

2 You can now create variables to hold the column names that will be used as nominal and analysis variable names. You can conditionally select rows and columns with the CASL .WHERE() operator and the [] selection syntax. For more information about row and column select, as well as the .WHERE() operator, see Chapter 11, “CASL Result Tables,” on page 85.

```
   targetVar =
"BAD";                                                    /*1*/
   inputCols =
{};                                                       /*2*/
   nominals =
{};                                                        /*3*/

   nominals = cardResult.Fetch.where(trim(_RLEVEL_)
                                     = 'CLASS')
[,'_VARNAME_'];           /*4*/

   inputCols = cardResult.Fetch.where(trim(_VARNAME_)
                                     != targetVar)
[,'_VARNAME_'];        /*5*/

   print
nominals;                                                          /*6*/
   run;
   print inputCols;
   run;
```

1 The Assignment statement creates the variable targetVar, which contains the target variable Bad. The target variable will be specified by the target parameter in the svmTrain action.

2 The Assignment statement creates the variable inputCols, which will contain the column names to use as analysis variables. InputCols will be specified by the inputs parameter in the svmTrain action set. The braces indicate an array.

3 The Assignment statement creates the CASL variable Nominals, which will contain the column names to use as nominal variables. InputCols will be specified by the inputs parameter in the svmTrain action set. The braces indicate an array.

4 Build the array of column names for the Nominal variable by using row and column selection. Row selection is done using the .WHERE() operator to subset the rows in the result table. The expression returns column names where the _RLEVEL_ value is CLASS. Column selection is done using the [ ] syntax. The code `[,'_VARNAME_']` specifies that the column _VARNAME_ is selected.

5 Build the array of column names for the inputCols variable by using row and column selection. Row selection is done using the .WHERE() operator to subset the rows in the result table. The expression returns column names where the _VARNAME_ value is not the target variable. Column selection is done using the [ ] syntax. The code `[,'_VARNAME_']` specifies that the column _VARNAME_ is selected.

6 The PRINT statement displays the contents of the Nominals and inputCols variables. Because the variables are arrays, they are written to the SAS log.

> **TIP** When arrays of strings are printed, they are displayed in the SAS log without quotation marks. You do not need to enclose the values in quotation marks when you supply the array to an action because CASL knows the beginning and ending of each string.

Output 10.9 *Listing of the Values in the Variables Nominal and InputCols*

```
       print nominals;
       run;
{BAD    ,REASON ,JOB     ,DEROG   ,DELINQ ,NINQ    }
       print inputCols;
       run;
 {LOAN    ,MORTDUE,VALUE   ,REASON ,JOB     ,YOJ     ,DEROG   ,DELINQ ,
  CLAGE   ,NINQ    ,CLNO    ,DEBTINC}
```

3 Run the svmTrain action to build the model. Use the variable InputCols as input for the inputs parameter. Use the Nominals variables as input to the nominals parameter. The RESULT= option specifies that the table is stored in the variable Svmresults. Now that the nominal and analysis column names are stored in the appropriate variable, you can use the variables as input to the inputs. You can use the nominals parameters in the svmTrain action.

```
   action svm.svmTrain /
      table='hmeq',
      inputs=inputCols,
      nominals=nominals,
      target=targetVar;
   run;
quit;
```

Output 10.10 Results: svmTrain Action

Results from svm.svmTrain

Model Information	
Task Type	C_CLAS
Optimization Technique	Interior Point
Scale	YES
Kernel Function	Linear
Penalty Method	C
Penalty Parameter	1
Maximum Iterations	25
Tolerance	1e-06

Number of Observations Read	5960
Number of Observations Used	3364

Training Results	
Inner Product of Weights	19.8000318
Bias	1.53729256
Total Slack (Constraint Violations)	532.92348
Norm of Longest Vector	2.72195233
Number of Support Vectors	3361
Number of Support Vectors on Margin	267
Maximum F	2.99999428
Minimum F	-1.0000874
Number of Effects	12
Columns in Data Matrix	49

Severity and Reason Codes

Severity Codes

A severity code indicates whether an action succeeded or failed. The following table lists the possible values and the meaning of each.

Table 10.1 Severity Codes

Severity Code	Severity Level Indicated	Description
0	Normal	Indicates that the action completed successfully.

Severity Code	Severity Level Indicated	Description
1	Warning	Indicates a minor issue that did not prevent the action from completing successfully. The reason code might provide additional information.
2	Error	Indicates an error that prevented the action from completing successfully. The reason code provides additional information.

Reason Codes

A reason code provides general information about the status code. The following table lists the possible values and the meaning of each.

Table 10.2 *Reason Codes*

Reason Code	Reason Category	Description
0	Success	Indicates that the action completed successfully.
1	Authorization	Indicates a permission problem.
2	Network	Indicates a network problem during connection or a network connection failure.
3	Memory	Indicates an insufficient memory condition.
4	Authentication	Indicates an authentication error.
5	Exception	Indicates an unexpected error condition that prevented the action from completing successfully.
6	Termination	Indicates a severe error condition that cause the action to terminate.

Program with Action Status Codes

The following example demonstrates how to work with the status that is returned by an action.

```
proc cas;
   session casauto;

   table.loadTable result=r status=sc /                    /* 1 */
      path="foo.csv"                                       /* 2 */
      casout={name="foo"};
   run;

   print "Severity code: " sc.severity;                    /* 3 */
   print "Reason:        " sc.reason;
   print "Status:        " sc.status;
   run;
   if (0 == sc.severity) then do;
      table.columnInfo / table="foo";
   end;
   run;
quit;
```

1 The action status is saved in a variable that is named Sc.

2 Trigger an error, by using the table.loadTable action to access a file that does not exist.

3 The PRINT statement writes the value for each of the keys in the status to the SAS log.

```
Severity:    2
Reason:      0
Status:      The action was not successful.
```

Key Ideas

- Actions are the building blocks of CASL programs. Actions are organized into groups of similar functionality called action sets.
- The action set name is optional, but an action name might not be unique among all action sets. It is best to include both the action set name and the action name to avoid ambiguity.
- Each action is a self-contained request. After creating a session, client programs makes a series of requests to invoke actions on that session. The actions of a particular session execute one at a time.
- Most actions return a dictionary that contains a result table. The result tables can then be manipulated, printed, or graphed.
- You can use the status codes returned by action for error checking.
- Use the DESCRIBE statement to view the structure of your results. Use the PRINT statement to print output the current open output destination. For more information, see Chapter 6, “Using the DESCRIBE Statement,” on page 35.

See Also

- Cloud Analytic Services Actions: A Holistic View
- “ACTION Statement” in *SAS Cloud Analytic Services: CASL Reference*
- *SAS Viya: System Programming Guide*
- *SAS Visual Analytics: Programming Guide*
- *SAS Visual Statistics: Programming Guide*
- *SAS Visual Data Mining and Machine Learning: Programming Guide*
- *SAS Visual Data Mining and Machine Learning: Deep Learning Programming Guide*
- *SAS Visual Text Analytics: Programming Guide*
- *SAS Optimization: Mathematical Optimization Programming Guide*
- *SAS Econometrics: Programming Guide*
- *SAS Visual Forecasting: Programming Guide*
- *SAS Data Quality: CAS Action Programming Guide*

11

CASL Result Tables

Overview

Result tables are typically summarizations of data or information.

Summary result table	Informational result table
An example of a summarization is the simple.summary action. When you run the simple.summary action, the action returns descriptive statistics for numeric variables such as the number of observations, number of missing values, mean value, and so on.	An example of an informational result table is the information that is returned by the decisionTree.dtreeTrain action. Informational result tables are common with actions that perform modeling.

Results from simple.summary

Descriptive Statistics for IRIS			
Column	N	Mean	N Miss
SepalLength	150	58.4333	0

Results from decisionTree.dtreeTrain

Decision Tree for IRIS	
Number of Tree Nodes	35.000000
Max Number of Branches	2.000000
Number of Levels	6.000000
Number of Leaves	18.000000
Number of Bins	20.000000
Minimum Size of Leaves	5.000000
Maximum Size of Leaves	20.000000
Number of Variables	2.000000
Alpha for Cost-Complexity Pruning	0
Number of Observations Used	150.000000
Maximum STD of Leaves	8.404761
Minimum STD of Leaves	0.330719
Mean Squared Error	9.701768

Operators

Operator	Definition	Syntax
.where()	Subsets the rows in a result table according to the expression. This operator returns a new result table.	*result-table*.**where**(*filter-expression*)
.compute()	Adds a temporary computed column to a result table. The values are computed according to an expression. This operator returns a new result table.	*result-table*.**compute**(<{'*variable-name*', '*label*', '*format*'},> *expression*)
[{*row-selection*}, {*column-selection*}]	Selects rows and columns from a result table. ■ Row selection is performed by position. You can specify a single row or an array of row positions. ■ Column selection is performed by position or by name. You can determine column names with the DESCRIBE statement. Depending on what you select, this operator can return a new result table, a dictionary, or an array. For more information, see "Selecting Rows" on page 89 and "Selecting Columns" on page 91.	*result-table*[{*row-1, row-2, ...*}] *result-table*[*row-lower-bound*:*row-upper-bound*] *result-table*[,*column-positions*] *result-table*[*row-positions*, {*column-1, column-2, ...*}] *result-table*[,{"*column-name-1*", "*column-name-2*", ...}]

Result Table Properties

A dictionary is associated with a result table to provide additional information about the table. The dictionary entries are identified in the following table:

Table 11.1 *Result Table Properties*

Property	Description	Syntax
nrows	This property contains the number of rows in the table.	*result-table*.nrows;

Property	Description	Syntax
ncols	This property contains the number of columns in the table.	*result-table*.ncols;
attrs	This property contains action-specific attributes for the table. This property is not added by all actions.	*result-table*.attrs;
name	This property contains the name of the table. In most cases, it is the same as the dictionary key that is used to access the table from the results.	*result-table*.name;
title	This property contains the title of the table.	*result-table*.title;

To experiment with the sample commands, you can run the following code to load a SAS data set, run the decisionTree.dtreeTrain action, and work with the result tables.

Example Code 11.1 *Display Result Table Properties*

```
proc cas;
    decisionTree.dtreeTrain result=dt /
      table="iris"
      inputs={"sepallength", "sepalwidth"}
      target="petallength";
run;

   /* Begin by printing the results. */
   print dt;

   /* Assign the ModelInfo result table to a variable. */
   tbl = dt.ModelInfo;
run;

/* At this point, you can run the sample commands. */
print "Result table name:  " tbl.name;
print "Result table title: " tbl.title;
print "Number of rows:     " tbl.nrows;
print "Number of columns:  " tbl.ncols;
print "Result attributes:  " tbl.attrs;
run;
```

Example Code 11.1 *Result Table Properties*

```
Result table name:  ModelInfo
Result table title: Decision Tree for IRIS
Number of rows:     13
Number of columns:  2
Result attributes:  {Action=dtreeTrain,Actionset=decisionTree,CreateTime=xxxxx}
```

Accessing Result Tables

When an action returns a result, the data type for the result is most often a dictionary with one or more result tables stored as values.

```
proc cas;
   simple.freq result=r / table="cars" inputs={"origin"};
   describe r;
run;
```

The following output is written to the SAS log and indicates the following:

- The data type for the variable that is named r is a dictionary.
- The dictionary has one key-value pair.
- The key is Frequency. The value is a result table.

```
dictionary ( 1 entries, 1 used);
 [Frequency] Table ( [3] Rows [5] columns
Column Names:
 [1] Column           [Analysis Variable] (varchar)
 [2] CharVar          [Character Value ] (varchar)
 [3] FmtVar           [Formatted Value ] (varchar)
 [4] Level            [Level           ] (int64) [BEST12.]
 [5] Frequency        [Frequency       ] (double) [BEST12.]
```

The result table can be accessed from the result by specifying the dictionary name and the key: `r.Frequency`. This notation is used throughout this section.

> **TIP** An alternative syntax to access the result table is r[“Frequency”]. This alternative syntax is necessary when a result table name includes a period or a space character.

Selecting Rows

Example Result Table from the Simple.Freq Action

Run the following statements in order to perform the row selection samples in this section.

```
proc cas;
   simple.freq result=r / table="cars" input="make";
run;
```

Select a Single Row

Note: When you select a single row from a table, the result is a dictionary.

Example Code 11.2 *Select the First Row*

```
   print r.Frequency[1];
run;
```

The PRINT statement shows the following in the SAS log:

```
{Column=Make,CharVar=Acura,FmtVar=Acura        ,Level=1,Frequency=7}
```

Example Code 11.3 *Select the Last Row*

```
   print r.Frequency[r.Frequency.nrows];
run;
```

TIP See table properties on page 87 for information about nrows.

The PRINT statement shows the following in the SAS log:

```
{Column=Make,CharVar=Volvo,FmtVar=Volvo        ,Level=38,Frequency=12}
```

Select a Range of Rows

```
   print r.Frequency[1:3];
run;
```

This example shows the range operator (:).

Analysis Variable	Character Value	Formatted Value	Level	Frequency
Make	Acura	Acura	1	7
Make	Audi	Audi	2	19
Make	BMW	BMW	3	20

Select the First, Third, and Fifth Rows

```
   print r.Frequency[{1, 3, 5}];
run;
```

Analysis Variable	Character Value	Formatted Value	Level	Frequency
Make	Acura	Acura	1	7
Make	BMW	BMW	3	20
Make	Cadillac	Cadillac	5	8

Select the Third and First Rows

```
   print r.Frequency[{3, 1}];
run;
```

Analysis Variable	Character Value	Formatted Value	Level	Frequency
Make	BMW	BMW	3	20
Make	Acura	Acura	1	7

Selecting Columns

Example Result Table from the Simple.Summary Action

Run the following statements in order to perform the column selection samples in this section.

```
proc cas;
   simple.summary result=r / table="cars";
run;
```

Select a Single Column

Note: When you select a single column, the result is an array. Each field in the array stores a row value from the selected column.

Example Code 11.4 *Select the First Column*

```
   print r.Summary[,1];
run;
```

The PRINT statement shows the following in the SAS log:

```
{MSRP       ,Invoice    ,EngineSize ,Cylinders  ,Horsepower ,MPG_City   ,
MPG_Highway,Weight     ,Wheelbase  ,Length     }
```

Select the First Three Columns by Position

```
   print r.Summary[,1:3];
run;
```

Analysis Variable	Minimum	Maximum
MSRP	10280	192465
Invoice	9875.00	173560
EngineSize	1.3000	8.3000
Cylinders	3.0000	12.0000
Horsepower	73.0000	500.00
MPG_City	10.0000	60.0000
MPG_Highway	12.0000	66.0000
Weight	1850.00	7190.00
Wheelbase	89.0000	144.00
Length	143.00	238.00

Select Three Columns by Name

Note: When you print a table, the column label is printed instead of the column name. You can determine the column names with the DESCRIBE statement.

```
   print r.Summary[,{"min", "max", "n"}];
run;
```

Minimum	Maximum	N
10280	192465	428
9875.00	173560	428
1.3000	8.3000	428
3.0000	12.0000	426
73.0000	500.00	428
10.0000	60.0000	428
12.0000	66.0000	428
1850.00	7190.00	428
89.0000	144.00	428
143.00	238.00	428

Select Four Columns by Position and Name

```
   print r.Summary[,{1, "min", "max", "mean"}];
run;
```

Analysis Variable	Minimum	Maximum	Mean
MSRP	10280	192465	32775
Invoice	9875.00	173560	30015
EngineSize	1.3000	8.3000	3.1967
Cylinders	3.0000	12.0000	5.8075
Horsepower	73.0000	500.00	215.89
MPG_City	10.0000	60.0000	20.0607
MPG_Highway	12.0000	66.0000	26.8435
Weight	1850.00	7190.00	3577.95
Wheelbase	89.0000	144.00	108.15
Length	143.00	238.00	186.36

Combining Row and Column Selection

Example Result Table from the Simple.CrossTab Action

Row and column selection can be combined. You can also use brackets and the .where() function.

Run the following statements in order to perform the row and column selection samples in this section.

```
proc cas;
   simple.crossTab result=r / table="cars" row="make" col="drivetrain";
run;

   print r.CrossTab[1:3];
run;

   describe r.CrossTab;
run;
```

The print statement prints the first three rows from the result table. The result table is named CrossTab.

Make	All	Front	Rear
Acura	1	5	1
Audi	12	7	0
BMW	5	0	15

The DESCRIBE statement prints the following information to the SAS log. Notice how the column names in the result table (Col1, Col2, Col3) correspond to the distinct values of the Drivetrain column.

```
[Crosstab] Table ( [38] Rows [4] columns
Column Names:
[1] Make             [                ] (varchar)
[2] Col1             [All             ] (double)
[3] Col2             [Front           ] (double)
[4] Col3             [Rear            ] (double)
```

Select Two Columns from the First Five Rows

```
print r.CrossTab[1:5,{"make", "col1"}];
```

```
run;
```

Make	All
Acura	1
Audi	12
BMW	5
Buick	1
Cadillac	1

Filter Rows with the Where Operator

```
   print r.CrossTab.where(col1 > 5);
run;
```

Make	All	Front	Rear
Audi	12	7	0
Mercedes-Benz	6	0	20
Subaru	11	0	0

Filter Rows and Subset the Columns by Position

```
   print r.CrossTab.where(col1 > 5)[,{1,3,4}];
run;
```

Note: You can filter rows by values of a column that are not included in the results. The rows are filtered on the number of all-wheel drive vehicles. However, that column is not selected with the bracket operator.

Make	Front	Rear
Audi	7	0
Mercedes-Benz	0	20
Subaru	0	0

Subset Rows by Position and Add a Computed Column

```
   print r.CrossTab.compute({"total", "Total"}, col1 + col2 + col3)
[1:5];
run;
```

Make	All	Front	Rear	Total
Acura	1	5	1	7
Audi	12	7	0	19
BMW	5	0	15	20
Buick	1	8	0	9
Cadillac	1	5	2	8

Iterating over Rows and Columns

Example Result Table from the Simple.Distinct Action

Run the following statements in order to perform the iteration samples in this section.

```
proc cas;
   simple.distinct result=r / table="cars";
run;
   describe r.Distinct;
run;
   print r.Distinct[1:5];
run;
```

The DESCRIBE statement shows the data structure of the result table. The information is shown in the SAS log.

```
[Distinct] Table ( [15] Rows [4] columns
Column Names:
[1] Column           [Analysis Variable] (string)
[2] NDistinct        [Number of Distinct Values] (double) [BEST12.]
[3] NMiss            [Number of Missing Values] (double) [BEST12.]
[4] Trunc            [Truncated       ] (double) [BEST3.]
```

Distinct Counts for CARS			
Analysis Variable	Number of Distinct Values	Number of Missing Values	Truncated
Make	38	0	0
Model	425	0	0
Type	6	0	0
Origin	3	0	0
DriveTrain	3	0	0

Iterate over Rows

- When you iterate over a result table, you iterate over the first dimension, the rows.
- When you access an individual row, the row is represented as a dictionary.

```
  do row over r.Distinct;
    print (row.Column || "    " || row.NDistinct);
  end;
run;
```

Because the PRINT statement is run iteratively on a dictionary, the output is written to the SAS log.

```
Make          38
Model         425
Type          6
Origin        3
DriveTrain    3
MSRP          410
Invoice       425
EngineSize    43
Cylinders     8
Horsepower    110
MPG_City      28
MPG_Highway   33
Weight        348
Wheelbase     40
Length        67
```

Access Rows as Key-Value Pairs

- The first DO loop is identical to the preceding example. This loop accesses each row as a dictionary.
- The inner DO loop iterates over the dictionary. Each column name is a key and enables you to access the column value.

```
  do row over r.Distinct[1:5];
```

```
    do k,v over row;
      if k in {'Column', 'NDistinct'} then do;
         print ('K=' || k || "   V=" || v);
      end;
    end;
  end;
run;
```

The PRINT statement shows the following in the SAS log:

```
K=Column   V=Make
K=NDistinct   V=38
K=Column   V=Model
K=NDistinct   V=425
K=Column   V=Type
K=NDistinct   V=6
K=Column   V=Origin
K=NDistinct   V=3
K=Column   V=DriveTrain
K=NDistinct   V=3
```

User-Defined Result Tables

You can define your own result table to combine results, status information, and so on, from several actions. Your result table can then be displayed on the **Results** tab in SAS. Other clients can also display result tables in client-specific ways.

The following code sample shows the two important functions for defining a result table.

- The NEWTABLE function enables you to create an empty result table.
- The ADDROW function enables you to use an array to append a row to the result table. Each field in the array corresponds to a column in the result table.

```
proc cas;
   columns = {"x", "y", "z"};
   coltypes={"integer", "double", "string"};
   table = newtable("X greater than A, Y greater than B", columns,
coltypes);

   do i = 1 to 5;
         z = (string)i;
         do j = 1 to 5;
              x = (string)j;
              row = {i, 2.6 * j, "abc" || x || z};
              addrow(table, row);
           end;
       end;
run;

   a = 3;
   b = 6;
```

```
   z = table.where((x>a)&&(y>b)).compute({"pct","Percent",best4.2},x/
y);
   print z;
run;
```

The PRINT statement shows the following in the SAS log:

z: Results

x	y	z	Percent
4	7.8	abc34	0.51
4	10.4	abc44	0.38
4	13	abc54	0.31
5	7.8	abc35	0.64
5	10.4	abc45	0.48
5	13	abc55	0.38

See Also

- See CASL function categories table for the functions that work with result tables.
- “NEWTABLE Function” in *SAS Cloud Analytic Services: CASL Reference*
- “ADDROW Function” in *SAS Cloud Analytic Services: CASL Reference*
- “DO OVER Statement” in *SAS Cloud Analytic Services: CASL Reference*
- “Simple Analytics Action Set” in *SAS Visual Analytics: Programming Guide*

12

CASL Missing Values

Definition

In CASL, if you reference an uninitialized variable the result is a missing value. By default, missing value is displayed as a single period (.), and a missing character value as a blank space.

missing value
: is a value that indicates that no data value is stored for the variable in the current observation.

By default, SAS prints a missing numeric value as a single period (.) and a missing character value as a blank space.

Numeric Missing Values

A missing numeric value is smaller than all numbers. If you sort your data set by numeric values, observations with missing values for a numeric variable appear first in the sorted data set. For numeric variables, you can compare special missing values with numbers and with each other. CASL supports 28 missing values. Missing values sort as the smallest values where ._ is the smallest and .Z is the largest. Missing values are supported only by numeric data with a DOUBLE data type.

The following table shows the sorting order of numeric values:

Table 12.1 Numeric Value Sort Order

Sort Order	Symbol	Description
smallest	._	underscore
	.	period
largest	.A-.Z	positive numbers

Any operation with a missing value results in a missing value. The following example demonstrates evaluating an expression with a missing value.

Example Code 12.1 *Evaluating Numeric Missing Values*

```
proc cas;
  x=15*.+8;
  print "x= "x;
run;
```

Example Code 12.1 *SAS Log*

```
x=.
```

Character Missing Values

Character missing values are smaller than any printable character value. When you sort a data set by a character variable, observations with missing (blank) values for the BY variable appear before observations in which values for the BY variable contain only printable characters. Some usually unprintable characters have values less than the blank. Therefore, when your data includes unprintable characters, missing values might not appear first in a sorted data set.

The following example demonstrates sorting a list with character missing values. The example contains an unprintable character (~), and other printable characters such as regular character values including blank space, .Z, and $.

Example Code 12.2 *Sorting Character Missing Values*

```
proc cas;
  Students={"Sylvester", "Angela", "Nikita", "~", "$", "Sumon",
"Kevin", " ", ".Z"};
  ClassAStudents=sort(Students);
  print "ClassAStudents= " ClassAStudents;
run;
```

The unprintable character (~) is not displayed first the list as it cannot be sorted. Therefore, it appears last.

Example Code 12.2 *SAS Log*

```
ClassAStudents= { ,$,.Z,Angela,Kevin,Nikita,Sumon,Sylvester,~}
```

13

Writing User-Defined Actions

Overview of User-Defined Actions

CASL server-side processing involves writing a CASL program that is stored on the CAS server. The stored CASL program is defined as a user-defined action set.

Because the action set is stored where the server can access it, the CASL statements can be written once and run by many users. This can reduce the need to exchange files between users that store common code.

The following list identifies some of the differences with server-side processing, compared to the conventional client/server programming that has been shown in this document:

- When your user-defined action runs a CAS action supplied by SAS, it is important to perform error checking.
- You need to use CASL functions such as SEND_RESPONSE that are used with server-side programming exclusively.
- Establish interface rules at your site. For example, you might adopt the rule that if an action experiences an error, only the status is returned. Similarly, if an action typically returns a result table, adopt the rule that the action always returns a result table—even if it has no rows due to filtering.

Note: You cannot add, remove, or modify a single user-defined action. You must redefine the entire action set.

Basic Workflow for User-Defined Actions

The workflow for server-side programming with CASL is as follows:

1. Most developers write a series of routines and functions that run successfully when submitted from the client.
2. The initially written code requires some modification to use CASL functions that are specific to server-side processing.
3. The builtins.defineActionSet action is used to add the code as your own action set and actions.
4. Store the user-defined action set as an in-memory table with the builtins.actionSetToTable action. Then, save the table with the table.save action.
5. Others who need to run the code use the builtins.actionSetFromTable action to make the user-defined action set available from their sessions.

User-Defined Action Example

Write the Routine

This code sample shows the following:

- A user-defined function that lists the file in a caslib with the table.fileInfo action.
- The result table is filtered by a user ID—the file system owner associated with the file.

The core functionality of this user-defined action—run the table.fileInfo action and filter the results—can be leveraged when writing a user-defined action.

Example Code 13.1 *List a User's Files: User-Defined Function*

```
function list_files(caslib_, owner_);
   table.fileInfo result=r / caslib=caslib_;
   ownedFiles = r.FileInfo.where(owner eq owner_);
   return ownedFiles;
end function;

print list_files("public", "sasdemo");
```

Modify the Code to Become a User-Defined Action

This code sample shows the following:

- The core functionality of the user-defined function is preserved.
- User-defined actions are grouped into action sets to organize functionality.
- The arguments to the user-defined function are modified to become parameters to the user-defined action. Type-checking for parameters is automatically performed by the server when the action is run.
- The SEND_RESPONSE function is the key piece that transfers the objects created on the server back to the client.

Example Code 13.2 *List a User's Files: User-Defined Action*

```
proc cas;
   builtins.defineActionSet /
      name="demo"                                                    /* 1 */
      actions={
      {                                                              /* 2 */
         name="listFiles"                                            /* 3 */
         desc="Return a list of all files owned by a user in a caslib"
         parms={                                                     /* 4 */
            {name="caslib" type="string" required=TRUE}
            {name="owner" type="string" required=TRUE}
         }
         definition="                                                /* 5 */
            table.fileInfo result=r status=s / caslib=caslib;
            if 0 != s.severity then do;                              /* 6 */
               send_response(s);
            end;
            searchFor = owner;                                       /* 7 */
            myTables = r.FileInfo.where(owner eq searchFor);
            resp.ListFiles = myTables;                               /* 8 */
            send_response(resp);
         "
      }
    }
  ;
run;

demo.listFiles result=lf / caslib="public" owner="sasdemo";
print lf.ListFiles;                                                  /* 8 */
run;
```

1 The name parameter is used to specify the entire action set name. Demo is used to reinforce this demonstration example.

2 The definition for each action is enclosed in braces. You can define several actions in a single call to the builtins.defineActionSet action.

3 The action name is set to ListFiles.

4 The parms parameter is used to specify the input parameters for the action. Two parameters, Caslib and Owner, are defined as required parameters with the

STRING data type. See For reference information about parameter data types, default values, and so on, see the builtins.defineActionSet action.

5 The definition parameter is a long body of code. These are the actions, CASL variables, CASL statements, and CASL functions that implement the user-defined action.

6 The status of the table.fileInfo action is stored in the dictionary that is named S. When the severity value is not 0, the action encountered an error. In the error condition, exit early and return the S dictionary with the EXIT function.

7 The searchFor variable stores the value of the Owner parameter. This enables the EQ operator to compare the Owner in the results with the Owner to use as a filter.

8 After filtering the result table from the table.fileInfo action, the dictionary variable Resp is created with a single key, ListFiles. The filtered result table is assigned to the key. The SEND_RESPONSE function returns the variable from the user-defined action to the client session.

9 The user-defined action, demo.listFiles is run and stores the results in a variable named Lf. Because the action stores the result table in the ListFiles key, the result table is accessed on the client from the Lf.ListFiles variable.

The PRINT statement shows the following in the SAS log:

```
NOTE: Added action set 'demo'.
{actionset=demo}
```

Note: Output for the demo.listFiles action is not shown because the information is site-dependent.

Persist the Action Set

- After the builtins.defineActionSet action is run, the user-defined action set exists in the current CAS session only.
- You must persist the action set to a SASHDAT file to make it reloadable, or you need to run the program with the builtins.defineActionSet code again.
- SASHDAT files can be saved to path-based caslibs only such as Path, DNFS, and S3.
- The following code sample uses the active caslib. Use a caslib with access controls that meets your needs. A personal caslib such as Casuser is limited to access by you only. A caslib such as Public is typically configured to provide access to all users.

```
builtins.actionSetToTable /                                    /* 1 */
   actionSet="demo"
   casOut={name="demo" replace=True};

table.save /
   table="demo"
   name="demoActionSet.sashdat"                                /* 2 */
```

```
      replace=True;
run;
```

1 The builtins.ActionSetToTable action makes an in-memory table from the user-defined action set in the active caslib.

2 The table.save action is used to persist the in-memory table to a SASHDAT file. You must use a path-based caslib type such as Path, DNFS, or S3.

The PRINT statement shows the following in the SAS log:

```
NOTE: Cloud Analytic Services saved the file demoActionSet.sashdat in caslib
      CASUSER(sasdemo).
{caslib=CASUSER(sasdemo),name=demoActionSet.sashdat}
```

Restore the Action Set from a Table

- The action set can be reconstructed in a session from the SASHDAT file that is used to persist it.
- Access controls can be applied to the SASHDAT file in the caslib to protect access to the code.

```
  builtins.actionSetFromTable /
    table="demoActionSet.sashdat"                              /* 1 */
    name="demo";

  demo.listFiles / caslib="public" owner="sasdemo";
run;
```

1 The action reads from an in-memory table. As an alternative to running table.loadTable first, you can specify the SASHDAT file name. The server temporarily loads the table as a transient-scope table, the action runs to restore the action set, and then the table is dropped automatically.

The PRINT statement shows the following in the SAS log:

```
NOTE: Added action set 'demo'.
{actionset=demo}
```

Note: Output for the demo.listFiles action is not shown because the information is site-dependent.

See Also

- For reference information about parameter data types, default values, and so on, see the builtins.defineActionSet action.

- For information about SEND_RESPONSE and related functions, see the server-side functions listed in the function categories table.
- For additional conceptual information related to server-side processing, see the details section of the CAS Server Action Set.