## Contents

**Chapter 1 • CAS Procedure** .......................... 1  
Overview: CAS Procedure ................................ 2  
Syntax: CAS Procedure .................................... 3  
Using: CAS Procedure ..................................... 27  
Results: CAS Procedure ................................... 40  
Examples: CAS Procedure ................................. 40  

*Recommended Reading* ................................. 55
# Chapter 1
## CAS Procedure

**Overview: CAS Procedure**
- About the CAS Procedure .......................................................... 2
- About the CAS Language ........................................................... 2
- Terminology ................................................................. 3

**Syntax: CAS Procedure** ......................................................... 3
- PROC CAS Statement ............................................................. 6
- ACTION Statement ............................................................... 6
- ASSIGNMENT Statement ........................................................ 8
- CALL Statement ................................................................. 9
- CONTINUE Statement ........................................................... 9
- DESCRIBE Statement ............................................................ 9
- DEPORT Statement ............................................................... 10
- DO Statement ................................................................. 11
- DO Statement, Iterative Statement ......................................... 12
- DO OVER Statement .......................................................... 13
- DO UNTIL Statement .......................................................... 14
- DO WHILE Statement ......................................................... 15
- END Statement ............................................................... 16
- FILE Statement ............................................................... 16
- FUNCTION Statement ....................................................... 17
- FUNCTIONLIST Statement ................................................... 17
- GOTO Statement .............................................................. 18
- IF-THEN/ELSE Statement .................................................... 18
- IMPORT Statement ............................................................ 19
- LEAVE Statement ............................................................. 20
- LOADACTIONSET Statement .................................................. 20
- ON Statement ............................................................... 20
- OUTPUT Statement .......................................................... 21
- PRINT Statement ............................................................. 22
- RAISE Statement ............................................................. 22
- RETURN Statement .......................................................... 23
- SAVERESULT Statement ...................................................... 23
- SELECT Statement ........................................................... 24
- SESSION Statement .......................................................... 25
- SOURCE Statement .......................................................... 26
- UNSET Statement ............................................................ 26

**Using: CAS Procedure** ......................................................... 27
- Arrays ................................................................. 27
- Dictionary ............................................................... 28
- Expression Parser ............................................................ 28
Overview: CAS Procedure

About the CAS Procedure

The CAS procedure enables you to interact with SAS Cloud Analytic Services (CAS) from the SAS client by providing you a programming environment based on the CASL language specification. The programming environment enables you to run CAS actions and use the results to prepare the parameters for another action.

The CAS procedure has several features that enable you to perform the following operations:

• Run any CAS action that is supported by the server, even if the action did not exist at the time of the release.

  Note: addTable action is currently not supported.

• Use multiple sessions to perform asynchronous execution.

• Operate on parameters and results as variables using the full function expression parser.

• Import your own executables that define callable functions.

About the CAS Language

• CASL is the language specification that enables you to access the CAS server. CASL is an integral part of the CAS procedure. It is designed to look and feel like the SAS language, such as DATA step, IML, and CMP.

• The language is case insensitive when referencing variables, either created by CASL or as a result of an action. You cannot define the data type for a variable in a variable declaration. The data type of a variable is determined by the type of value that is assigned to it.

• The values that are in CASL are CAS values that enable you to pass the values as parameters to the CAS server. CASL also supports stand-alone arrays and dictionaries.
**Terminology**

**action**
A task that is performed by the server at the request of the user. An action sends a request to the server, which parses the arguments of the request and invokes the action function. The result is returned and resources are cleaned.

**action set**
A collection of actions (tasks) that group functionality such as session management, table management, and so on.

**argument**
A value that is supplied to a procedure when it is called to perform an operation.

**condition**
One or more numeric or character expressions that result in a value on which some decision depends.

**expression**
An expression is used in program statements to create, assign, calculate, and transform new values and to perform conditional processing.

**function**
Is a group of statements that perform a task together. A function takes a variable and a parameter and returns a value.

**results table**
A table is created as the result of an action, and can be used for other actions. In addition to rows and columns, the results table also contains labels, attributes, and variable types.

**CAS session**
A session represents a user that has logged onto the server. The session can then be used to submit actions and produce results.

**variable**
A symbolic name for a value. The value can be a list, a dictionary, or a simple data types (string, integer, or floating-point number). You can assign a value throughout a program.

---

**Syntax: CAS Procedure**

```cas
PROC CAS <exe> <noqueue>;
  ACTION <action-set-name.action-name><RESULT=<variable><STATUS=<rc><ASYNC=name>
  /<parameters>;
  ASSIGNMENT target = expression;
  CALL function (argument1, argument2...);
  CONTINUE;
  DESCRIBE variable;
  DEPORT <extension name>;
  DO;
    ... more CASL statements ...;
```
END;
DO UNTIL
  condition;
  ... more CASL statements ...;
END;
DO WHILE condition;
  ... more CASL statements ...;
END;
DO [<key,>] <var> OVER <value>;
  ... more CASL statements ...;
END;
DO <var> = <start> [TO <stop>] [BY <increment>] [
  WHILE <condition> | UNTIL <condition>];
  ... more CASL statements ...;
END;
FILE;
FUNCTION function name ( [argument 1 [, argument 2...]] );
  ... more CASL statements ...;
END;
FUNCTIONLIST <name>;
  FLIST <name>;
GOTO label;
IF expression
  THEN statement;
  ELSE <statement>;
IMPORT <extension name>;
LEAVE;
LOADACTIONSET actionsetname;
ON condition response expression;
OUTPUT libname;
PRINT expression;
RAISE condition-name;
RETURN <expression>;
SAVERESULT variable-name <noreplace> <dataout=<libref.> member-name> | <lib=libref> | <file = <path-name> filename >|<caslib=<caslib>>|<casout=name>;
SELECT select-expression; <when-list> ... <when-list>; OTHERWISE statement-list;
END end-label; <when-list> :: = WHEN <when-expression> <statement-list>;
SESSION session-name;
SOURCE variable; text; endsource;
UNSET <DISP> <LOGS>;
RUN;
QUIT;
<table>
<thead>
<tr>
<th>Statement</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC CAS</td>
<td>Enables you to interact with the CAS server from SAS using the CAS language and CAS actions.</td>
</tr>
<tr>
<td>ACTION</td>
<td>Runs SAS Cloud Analytic Services actions.</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>Assigns a value to a variable.</td>
</tr>
<tr>
<td>CALL</td>
<td>Calls function with the specified arguments.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Enables the next iteration of the loop to process without skipping any code in between.</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>Displays the data type of a variable created by the ACTION statement.</td>
</tr>
<tr>
<td>DO</td>
<td>Creates a block of code that executes as one statement.</td>
</tr>
<tr>
<td>DO UNTIL</td>
<td>Executes statements in a DO loop repetitively until a condition is true.</td>
</tr>
<tr>
<td>DO WHILE</td>
<td>Executes statements in a DO loop repetitively until a condition is true.</td>
</tr>
<tr>
<td>DO OVER</td>
<td>Iterates over a list or result table.</td>
</tr>
<tr>
<td>DO</td>
<td>Executes statements in a DO loop repetitively until a condition is true.</td>
</tr>
<tr>
<td>END</td>
<td>Ends a DO group or SELECT group processing.</td>
</tr>
<tr>
<td>FILE</td>
<td>Specifies different locations of output.</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>Creates a new function that can be called in an expression.</td>
</tr>
<tr>
<td>GOTO</td>
<td>Directs program execution immediately to the statement label that is specified.</td>
</tr>
<tr>
<td>IF-THEN/ELSE</td>
<td>Executes a SAS statement for observations that meet specific conditions.</td>
</tr>
<tr>
<td>FUNCTIONLIST</td>
<td>Loads the function in the specified extension and adds it to the list of available functions.</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Stops processing the current loop and resumes with the next statement in the sequence.</td>
</tr>
<tr>
<td>LOADACTIONSET</td>
<td>Loads a SAS Cloud Analytic Services action set. Some actions are available as platform-level functionality. You can use the LOADACTIONSET statement to load more action sets into your session.</td>
</tr>
<tr>
<td>ON</td>
<td>Enables you to specify how to handle an error during execution.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Writes the current observation to a SAS data set.</td>
</tr>
</tbody>
</table>
### Statement | Task
--- | ---
PRINT | Displays the value of the expression to the current output location.
RAISE | Sends a signal to the calling process or thread.
RETURN | Returns a value from the current function.
SAVERESULT | Creates a SAS data set from the results of an ACTION.
SELECT | Executes one of several statements or groups of statements.
SESSION | Specifies an existing session to use for running a program.
SOURCE | Embeds text in the program and assigns the program to the given variable.
UNSET | Controls message suppression in the SAS log.

---

**PROC CAS Statement**

Enables you to program and schedule SAS Cloud Analytic Services actions from the SAS client.

**Syntax**

```
PROC CAS <exc> <noqueue>;
```

**Optional Arguments**

- **exc**
  - Executes the CASL code as soon as the previous block of code has completed processing. The default option does not execute CASL code until a RUN statement is entered.

- **noqueue**
  - Forces output to be displayed as soon as output is produced.

**Details**

*Note:* Global statements, SAS macro code, and RUN statements do not terminate the procedure.

---

**ACTION Statement**

Runs SAS Cloud Analytic Services actions.

*Note:* You can use expressions to build up parameters for ACTION statements. To use an expression in a statement, wrap the expression in a dollar sign ($), followed by a parentheses ( ).
See: For documentation about the actions that you can run with the ACTION statement, see: SAS Viya: System Programming Guide, and SAS Visual Analytics Programming Guide.

Example: “Example 2: Run an Action” on page 41

Syntax

\[
\text{\langle ACTION\rangle \ action-set-name.\ action-name \ RESULT= \langle variable\rangle \ STATUS= \langle rc\rangle \ ASYNC= \langle name\rangle \ / \ \langle parameters\rangle;}
\]

Required Argument

\[\text{\langle action-set-name.\ action-name \}}\]

\[\text{specifies the action to run.} \]

\[\text{action-set-name} \]

\[\text{specifies the name of the action set that contains the action to run. Examples are tables, simple, and network.} \]

\[\text{action-name} \]

\[\text{specifies the name of the action to run. Examples are loadTable, listActions, and serverStatus.} \]

Example For example, the following statement specifies the loadtable action that is part of the tables action set:

\[
\text{action tables.loadtable / path="cholesterol.csv"; run;}\]

In this example, \text{PATH=} is a parameter of the loadtable action.

Optional Arguments

\[\text{parameter} \]

\[\text{specifies the specified action’s parameters. You can list the parameters for an action with the help action.} \]

Example action help submit / action="action-name"; run;

\[\text{RESULT=variable-name} \]

\[\text{stores the results of the action in a variable. You can then use the variable in other CASL statements and actions.} \]

Alias \text{R=}

See For examples of using results in a program, see “Working with Results and Creating Variables” in SAS Viya: System Programming Guide.

\[\text{STATUS= \langle variable-name\rangle} \]

\[\text{stores the status code of the action in a variable. You can use the variable with statements such as DESCRIBE and with a selected set of SAS statements within the scope of the CAS procedure invocation. If you do not specify a \text{variable-name}, a variable named \text{_status} that contains the status codes is created.} \]
ASSIGNMENT Statement

Evaluates an expression and stores the result in a variable.

**Tip:** If the variable already exists, the new assignment replaces the variable and the old value is overwritten.

**Syntax**

```
<ASSIGNMENT> variable = expression;
```

**Actions**

- **variable**
  - names a new or existing variable.

- **expression**
  - specifies the expression used in program statements to create variables, assign values, calculate new values, transform values, and perform conditional processing.

  **Tip** expressions can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign.

**Details**

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

**Example: Valid Assignment Statements**

This assignment statement defines the target variable as “numeric”.

```
x = "numeric";
```

This assignment statement defines the target variable as a numeric value of 88.

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

This target statement defines the target variable as $z$.

```
z = min(y, 70);
```
CALL Statement
Calls a function with the specified argument. If the function returns a value, it is ignored.

Syntax

\[ \text{CALL} \text{ function argument-1 argument-2, \ldots}; \]

Optional Arguments

- **function**: processes one or more arguments and can be used in either an assignment statement or an expression.
- **argument**: specifies a value that is supplied to a procedure when it is called to perform an operation.

CONTINUE Statement
Enables the next iteration of the loop to process without skipping any code in between.

Syntax

\[ \text{CONTINUE}; \]

DESCRIBE Statement
Displays the contents of a variable created by the ACTION statement.

Syntax

\[ \text{DESCRIBE variable}; \]

Required Argument

- **variable**: specifies the name of a variable created by the ACTION statement.

See “ACTION Statement” on page 6

Examples

**Example 1: Using the DESCRIBE Statement**
The following example stores the result of the serverStatus action in the variable named \( r \).
The DESCRIBE statement displays the result as a list that includes three entries: about (a list), server (a table), and nodestatus (a table).

You can access the variable r and learn about the server.

**Example 2: Producing Results Using Previously Assigned Variables**

In the previous example, you stored the results in the variable r and received three entries. You can take one of the entries and print the data.

```plaintext
print r.server [1];
run;
```

The output gives you information about the CAS server.

```
[ nodes=143 , actions=17 ]
```

---

**DEPORT Statement**

Unloads the functions previously loaded with the IMPORT statement.

**Syntax**

```
DEPORT <extension-name>;
```
**Optional Argument**

extension- name

specifies the name of the import extension.

**Example: Unloading the Caslext Extension**

Use the DEPORT statement to unload the caslext extension that you imported using the IMPORT statement.

```plaintext
proc cas;
    deport caslext;
run;
```

---

**DO Statement**

Creates a block of code that executes as one statement.

**Requirement:** An END statement must follow a DO statement and all of the DO group processing statements.

**Example:** “Example 3: Define a New Function” on page 42

**Syntax**

```plaintext
DO;
    ... more CASL statements ...;
END;
```

**Details**

The DO statement is the simplest form of DO group processing. The statements between DO and END statements are called a DO group. You can nest DO statements within DO groups. The statements between the DO and END statements are called a DO group. A simple DO statement is often used within an IF-THEN/ELSE statement. See “IF-THEN/ELSE Statement” on page 18. A DO statement is executed depending on whether the IF condition is true or false.

**Example: Using the DO Statement**

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 7. If YEARS is less than or equal to 7, statements in the DO group do not execute, and the program continues with the assignment statement that follows the ELSE statement.

```plaintext
if years>7 then
do;
    months=years*12;
    put years= months=;
end;
else do;
    yrsleft=7-years;
end;
```
DO Statement, Iterative Statement

Iterates over a list.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Tips: The order of the optional TO and BY clauses can be reversed. When you use more than one specification, each one is evaluated before its execution.

Example: “Example 6: Computing Partial Dependency” on page 48

Syntax

```
DO <variable> = <start> TO <stop>
<BY increment> <WHILE (condition) | UNTIL (condition)>;
    … more CASL statements …;
END;
```

Optional Arguments

- **variable**
  names a variable whose value governs execution of the DO group.

- **start**
  specifies the initial value of the variable.

  When both start and stop are present, execution continues (based on the value of increment) until the value of index-variable passes the value of stop. When only start and increment are present, execution continues (based on the value of increment) until a statement directs execution out of the loop, or until a WHILE or UNTIL expression that is specified in the DO statement is satisfied. If neither stop nor increment is specified, the group executes according to the value of start. The value of stop is evaluated before the first execution of the loop.

- **stop**
  specifies the ending value of the index variable.

- **BY increment**
  specifies a positive or negative number (or an expression that yields a number) to control the incrementing of index-variable.

  The value of increment is evaluated before the execution of the loop. Any changes to the increment that are made within the DO group do not affect the number of iterations. If no increment is specified, the index variable is increased by 1. When increment is positive, start must be the lower bound and stop, if present, must be the upper bound for the loop. If increment is negative, start must be the upper bound and stop, if present, must be the lower bound for the loop.

- **WHILE (condition) | UNTIL (condition)**
  evaluates, either before or after execution of the DO group, any condition that you specify. Enclose the condition in parentheses.

  A WHILE expression is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. An
UNTIL expression is evaluated after each execution of the loop, so that the statements inside the group are executed repetitively until the expression is true.

Restriction: A WHILE or UNTIL specification affects only the last item in the clause in which it is located.

Details

There are four other forms of the DO statement:

- The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The DO OVER statement iterates over a list.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Example: Using the Do Loop Statement

This statement identifies the loop variable as $i$.

```
do i=1 to 10;
```

This statement executes the loop variable in increments of 3.

```
do i=1 to 10 by 3;
do i=1 to 10 while( i < 5);
do i=1 to 10 by 1 until ( i>=5);
```

DO OVER Statement

Iterates over a list.

**Requirement:** An END statement must follow a DO statement and all of the DO group processing statements.

**Example:** "Example 4: Use Run-time Math Functions" on page 43

**Syntax**

```
DO <key>, <var> OVER <value>;
... more CASL statements ...;
END;
```

**Optional Arguments**

- **key** specifies the key or index for the list.
- **var** specifies the variable contexts.
- **value** specifies the list.
Example: Using the DO OVER Statement

The following example takes the results of an action, iterates over the results to find Nmiss values, and then saves the values in a list. That list can then be used as input to another action. For the complete example, see “Example” in SAS Viya: System Programming Guide.

```
action simple.summary result=CPSSum                      /* 1 */
   table={caslib="casdata", name="phoneSubs"];           /* 1 */
run;

cinfo = findTable(CPSSum);                               /* 2 */
   nmissVar = {*Country Name", *Indicator Name*};        /* 2 */
   do col over cInfo;                                    /* 3 */
      if (col.NMiss <20) then do;                        /* 4 */
         nmissVar= nmissVar + col.Column;               /* 4 */
      end;
   end;
   print nmissVar;                                       /* 4 */
run;
```

1 The Summary action creates descriptive statistics such as sum, means, and nmiss. The RESULT= option saves the results in a variable named CPSSum.

2 Cinfo is the name of the variable that contains the results of the FINDTABLE function. FINDTABLE is an internal function that searches for a value in a table. CPSSum is the variable that holds the results of the Summary action.

3 NmissVar is the name of the variable that will hold the results of the DO OVER statement. The brackets { } indicate that a list is being made. Country Name and Indicator Name are columns to be included in the list. The rest of the columns are added by the DO OVER statement.

4 The DO OVER statement iterates over the results stored in the variable cInfo. Col is the name of the index for the list.

5 The IF THEN DO statement finds columns that have a value of NMiss greater than twenty.

6 The columns are then added to the list in the variable nmissVar.

7 The PRINT statement prints the variable nmissVar to the SAS log.

**Output 1.1 SAS Log**

```
```

**DO UNTIL Statement**

Executes statements in a DO loop repetitively until a condition is true.

**Requirement:** An END statement must follow a DO statement and all of the DO group processing statements.
Syntax

DO UNTIL condition;
    ... more CASL statements ...;
END;

Required Argument

condition
    specifies one or more numeric or character expressions that result in a value on
which some decision depends.

Example: Using a DO UNTIL Statement to Repeat a Loop

This statement repeats the loop until \( x \) is greater than 10. The expression \( x > 10 \) is evaluated at
the bottom of the loop. There are ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

\[
x = 0;
do until (x > 10);
    print x;
    x = x + 1;
end;
\]

DO WHILE Statement

Executes statements in a DO loop repetitively while a condition is true.

Restriction:  An END statement must follow a DO statement and all of the DO group processing
statements.

See:  "DO Statement, Iterative Statement" on page 12

Syntax

DO WHILE condition;
    ... more CASL statements ...;
END;

Required Argument

condition
    specifies one or more numeric or character expressions that result in a value on
which some decision depends.

Example: Using a DO WHILE Statement

These statements repeat the loop while \( x \) is greater than 5. The expression \( x < 10 \) is evaluated at
the top of the loop. There are ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

\[
x = 0;
do while (x < 10);
    print x;
    x = x + 1;
end;
\]
END Statement

Ends a DO group, FUNCTION, or SELECT group processing.

**Requirement:** The END statement must be the last statement in a DO group, FUNCTION, or a SELECT group.

**Syntax**

```
END;
```

**Without Arguments**

Use the END statement to end DO group, FUNCTION, or SELECT group processing.

FILE Statement

Enables you to specify different locations in the output.

**Example:** “Example 5: Train a Decision Tree Model and Score Data” on page 44

**Syntax**

```
FILE;
```

**Required Argument**

```
FILE
```

specifies to bypass ODS for CAS result tables and send output to the desired location.

**Examples**

**Example 1**

```
file tablefile "u/user/table.lst";
print table;
```

**Example 2**

Sets the active output location to ODS.

```
file ods;
```

**Example 3**

Sets the active output location to the file.

```
file nodes "/u/user/nodes";
action listnodes result=r submit;
```
The result of the listnodes action is printed to `/u/user/nodes`.

```
print r;
```

**Example 4**
Set the active output location to the SAS log.

```
file log;
```

The same results are printed to the log.

```
print r;
```

---

**FUNCTION Statement**

Creates a new function that can be called in an expression.

**Requirement:** An END statement must follow a FUNCTION statement.

**Example:** "Example 3: Define a New Function" on page 42

**Syntax**

```
FUNCTION function-name( argument-1 , argument-2 );
  statements;
END;
```

**Required Arguments**

- `function-name` specifies a user-defined function name.
- `argument` specifies the value that is supplied to a procedure when it is called to perform an operation.
- `statement` is any CASL statement.

---

**FUNCTIONLIST Statement**

Prints a list of available functions.

**Syntax**

```
FUNCTIONLIST <name>;
FLIST <name>;
```

**Optional Argument**

- `name` specifies the name of the available functions. The optional name qualifies the list.
GOTO Statement
Directs program execution to the specified statement label.

Syntax
GOTO label;

Required Argument
label
specifies a statement label that identifies the GO TO destination.

IF-THEN/ELSE Statement
Executes a SAS statement for observations that meet specific conditions.

Tip: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

Example: “Example 3: Define a New Function” on page 42

Syntax
IF expression THEN statement;
<ELSE ><statement>;

Required Arguments
expression
Used in program statements to create variables, assign variables, calculate new values, transform variables, and perform conditional processing.

statement
Any CASL statement.

Details
CAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing. A nonzero and nonmissing result causes the expression to be true. A result of zero or missing causes the expression to be false.

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a statement for observations that are read from a table. An optional ELSE statement gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement. Using IF-THEN statements without the ELSE statement causes CAS to evaluate all IF-THEN statements.

Using IF-THEN statements with the ELSE statement causes CAS to execute IF-THEN statements until it encounters the first true statement. Subsequent IF-THEN statements are not evaluated.
Example

```plaintext
x=5
if (x > 6) then put "x is greater than 6";
else print "x is less than or equal to six";
print "x=" x;
run;
```

**IMPORT Statement**

Loads the function in the specified extension and adds it to the list of available functions.

**Syntax**

```
IMPORT <extension-name>;
```

**Optional Argument**

*extension-name*

specifies the name of the import extension.

**Example: Import Caslex Extension**

The IMPORT statement loads the *caslex* extension and adds it to the list of available functions. The *caslex* extension contains two functions: `sumi` and `sumid`.

```plaintext
proc cas;
import caslex;
flist caslex;
run;
```

The SAS log includes the following message:

```
NOTE: Import Functions
NOTE: [1] sumi  : int64_t sumi(value,...);
NOTE: [2] sumid : double sumd (value,...);
```

Print each function value. The difference between the two functions is the return type, which is given in the description of the function.

```plaintext
x=sumi(1, 2.6, 2.7, 5);
   print x;
run;
```

The SAS log includes the following message:

```
11
```

```plaintext
x=sumd(1, 2.6, 2.7, 5);
   print x;
run;
```
The SAS log includes the following message:

11.3

LEAVE Statement

Stops processing the current loop and resumes with the next statement in the sequence.

**Tip:** You can use the LEAVE statement to exit a DO loop or SELECT group prematurely based on a condition.

**Syntax**

LEAVE;

**Without Arguments**

The LEAVE statement stops the processing of the current DO loop or SELECT group and continues processing with the next statement following the DO loop or SELECT group.

LOADACTIONSET Statement

Loads a SAS Cloud Analytic Services action set. Some actions are available as platform-level functionality. You can use the LOADACTIONSET statement to load more action sets into your session.

**Requirement:** The ability to load action sets into your session is subject to access controls.

**Example:** “Example 5: Train a Decision Tree Model and Score Data” on page 44

**Syntax**

LOADACTIONSET "action-set-name";

**Required Argument**

*action-set-name*

specifies the name of a SAS Cloud Analytic Services action set.

ON Statement

Enables you to specify error handling during execution.

**Syntax**

ON condition-response expression;
Required Arguments

condition-response
specifies an expression that evaluates an error and produces a value.

expression
specifies an expression is used in program statements to create variables, to assign values, to calculate new values, to transform variables, and to perform conditional processing.

Optional Arguments
The supported conditions are as follows:

unknown
This condition is raised after accessing a variable with an unknown value.

ilop
This condition is raised after performing an invalid operation.

Example
The following example demonstrates registering a function named Gety. In the function, the ON statement is used to register a response for handling unknown variables—in this case, to replace an unknown value with a replacement value, 3.2. Because the z variable is referenced, but undefined, the response handler for unknown values is triggered.

```proc cas;
repval = 3.2;
function gety(x) do;
on unknown value=repval response=replace;
y=z;
return (y);
end func;
x = 1;
print "y value = " get y(x);
run;
```

The SAS log includes the following message:

```
y value = 3.2
```

OUTPUT Statement
Sets the active output location.

Example:  "Example 6: Computing Partial Dependency" on page 48

Syntax

OUTPUT <ODS | LOG>;
**Required Arguments**

**ODS**
- specifies that the output is sent to an ODS destination.

**LOG**
- specifies that the output is sent to the SAS log.

---

**PRINT Statement**
Displays the value of the expression to the current output location.

**Example:**  “Example 4: Use Run-time Math Functions” on page 43

---

**Syntax**

`PRINT <expression> <expression>;`

---

**Optional Argument**

**expression**
- specifies an expression with no more than one *operator*. An expression can consist of one of the following single operators:
  - constant
  - variable
  - function

**Tip** You can include more than one expression in your PRINT statement.

---

**Example**

This example prints the variable r. The variable has to be previously defined.

```
print r;
```

---

**RAISE Statement**
Sends a signal to the calling process.

**Tip:** The following statement returns a 0 on success and a nonzero on failure: `raise ( );`.

---

**Syntax**

`RAISE condition;`

---

**Required Argument**

**condition**
- specifies one or more numeric or character expressions that result in a value on which some decision depends.
**RETURN Statement**

Returns a value from the current function.

**Example:**  “Example 5: Train a Decision Tree Model and Score Data” on page 44

**Syntax**

```
RETURN <expression>;
```

**Optional Argument**

`expression`

Used in program statements to create variables, assign variables, calculate new values, transform variables, and perform conditional processing.

**Details**

The RETURN statement can be used in conjunction with the “IF-THEN/ELSE Statement” on page 18 and the “FUNCTION Statement” on page 17.

---

**SAVERESULT Statement**

Creates a SAS data set from the results of an ACTION.

**Note:** A column that contains more than 32,000 characters is too large to save in a data set.

**Example:**  “Example 5: Train a Decision Tree Model and Score Data” on page 44

**Syntax**

```
SAVERESULT variable-name <NOREPLACE>
<DATAOUT=<libref.> data-set-name> | <LIB=libref> |
<FILE=file-specification>;
```

**Required Argument**

`variable-name`

includes one table result or a list of table results

**Optional Arguments**

`NOREPLACE`

specifies not to overwrite saved files.

`DATAOUT=<libref.>data-set-name`

specifies the SAS library and data set name.

`LIB=libref`

specifies which SAS library to use.
FILE=\textit{path-name} | \textit{filename} | CALIB=\textit{casref} | CALIB=\textit{casref}

specifies the file or filename to use.

\textit{path-name}

specifies the file path.

\textit{filename}

specifies the file name.

CALIB=\textit{<casref>}

specifies which caslib to use.

CASOUT=\textit{name}

specifies the name of the result table.

\section*{SELECT Statement}

Executes one of several statements or groups of statements.

\textbf{Restriction:} An END statement must follow a SELECT statement.

\textbf{Note:} SELECT statements can be nested.

\section*{Syntax}

\begin{verbatim}
SELECT <select-expression> ;
<when-list> ... <when-list> ;
OTHERWISE statement-list ;
END end-label ;
<when-list> ::= WHEN <when-expression> <statement-list> ;
\end{verbatim}

\section*{Required Arguments}

\textit{select-expression}

specifies an expression that evaluates to a single value.

\textit{end-label}

the END statement closes the SELECT statement. The optional end-label argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the SELECT statement, or an error will occur.

\textit{statement-list}

specifies any executable SAS statement, including DO, SELECT, and null statements. You must specify the \textit{statement} argument.

\textit{when-expression}

specifies any SAS expression, including a compound expression.

\textbf{Note:} You must specify at least one when-expression.

\textit{when-list}

can be any SAS compound expression containing at least one when-expression.
Details

**Using WHEN Statements in a SELECT Group**
The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify CAS statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

**Example: SELECT with a select-expression**
This example illustrates how to use the SELECT statement with a `select-expression`.

```plaintext
select (a);
when (1) x=x*6;
when (2);
when (3,4,5) x=x*20;
otherwise;
end;
```

**SESSION Statement**
Specifies an existing session to use for running a program.

**Example:**
“Example 5: Train a Decision Tree Model and Score Data” on page 44

**Syntax**
Form 1: `SESSION session-name;`
Form 2: `SESSION (“session-name”);`

**Required Argument**
`session-name`
specifies the name of an existing session.

**Examples**

**Example 1: Print Session ID**
This example prints your session ID for the `casauto` session.

```plaintext
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
cas casauto;
proc cas;
    session casauto;
    action sessionid result=uuid;
    print uuid;
run;

{CASAUTO:Tue Sep 13 11:16:18 2016=4ebd26ad-9dc4-cf4f-a108-4f497d06a23f}
```
Example 2: List Session Options
This example lists all the options for the casauto session.

```
cas casauto LISTSESSOPTS;
```

Example 3: Connect to an Existing Session
This example connects to an existing session using the UUID. UUID is session specific.

```
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
cas casauto uuid="4ebd26ad-9dc4-cf4f-a108-4f497d06a23f";
proc cas;
    session casauto;
    session.sessionstatus result=s;
    put s;
run;
```

Example 4: Create a Session Variable
You can use the SESSION statement to create a session variable. A session variable is best used with parallel sessions where you do not specify the name of the session. The parentheses is required for casauto to be the session variable.

```
proc cas;
    session ("casauto");
run;
```

SOURCE Statement
Enables you to embed text in the program and assign it to a given variable.

Syntax

```
SOURCE variable;
<text>;
ENDSOURCE;
```

Required Argument

```
variable
```

specifies a symbolic name for a value. The value can be a list, a dictionary, or a simple data types (string, integer, or floating-point number). You can assign a value and you can assign new values throughout a program.

Optional Argument

```
text
```

specifies a string representation of the value for the variable.

UNSET Statement
Controls message suppression in the SAS log.
Tip: You can specify the following statement to suppress both types of messages:

```sql
unset disp logs;
```

## Syntax

UNSET <DISP> <LOGS>;

### Optional Arguments

- **DISP**
  - Specifies to suppress printing the disposition in the SAS log.

- **LOGS**
  - Specifies to suppress printing messages from CAS in the SAS log.

## Example

The following SAS log demonstrates how the statement `unset disp;` is used to suppress the disposition from the SAS log:

```sql
60 proc cas;
61 session mysess;
62
63 setsessopt / caslib="myCasLib";
64 run;
NOTE: Active Session now mysess.
WARNING: The value myCasLib for casLib is invalid.
ERROR: The CAS server stopped processing this action because of errors.
Disposition: Severity Error
65 /* suppress the "Disposition:" message in the SAS log */
66 unset disp;
67 setsessopt / caslib="myCasLib";
68 run;
WARNING: The value myCasLib for casLib is invalid.
ERROR: The CAS server stopped processing this action because of errors.
```

---

### Using: CAS Procedure

#### Arrays

An array is a list of values. An array is created when you assign a value to one of the elements in the array. The number of items in the array is set by the highest index. All indexes in the array start at 1 and all the values do not have a key. Values that are not set are uninitialized. Elements in an array are referenced with brackets `[]`.

There are no limits to the number of dimensions that you can have for an array. For example, a two-dimensional array is a list of lists.

This is an example of an array:

```sql
sample[4,5,6] = 14
```
This example creates a three-dimensional array. The third dimension has the highest index of 4.

\[
\text{arrayex} = x[2,3,4]
\]

This is an example of an uninitialized value. The array index can be a normal expression within CASL. This example would result in an error.

\[
\text{box}[3*4+5, \text{box}, \text{function}(4,5,6)] = 15;
\]

**Note:** Numeric values, including constants, are always double unless explicitly created as int64 values.

**Dictionary**

A dictionary is a list of values that have a key name with a value. Dictionaries can be multidimensional.

Here are the two acceptable syntaxes for dictionaries:

\[
\text{a.foo.bar}=1;
\]

\[
\text{a["foo","bar"]}=1;
\]

**Expression Parser**

**Expression Evaluation**

The CASL language supports a full featured expression evaluation system. Each variable is a location to hold values and type information goes with the values. The result of an expression evaluation is a value. The value can be assigned to a variable. These are the rules for expression evaluation:

- \( \text{x} = 5 + 5 \); is an integer.
- \( \text{x} = 5 + 5.0 \); is a double.
- \( \text{x} = "5" + 5 \); is a double.
- \( \text{x} = (\text{int64}) "5" + 5 \); is an integer.
- \( 6/2 \); is a double.

**Note:** Numeric values, including constants, are always doubles unless explicitly created as int64 values.

**Variable and Literal Mode**

By default the expression parser treats all names as variable names. A value must be enclosed in quotation marks to be a literal string of the name.

\[
\text{model} = "\text{MODEL}";
\]

\[
\text{make} = "\text{MAKE}"
\]

\[
\text{mpg} = "\text{MPG}"
\]
Operators in Expressions

Definitions
A SAS operator is a symbol that represents a comparison, arithmetic calculation, or logical operator; a SAS function; or group parentheses. SAS uses two major types of operators:

- prefix operators
- infix operators

A prefix operator is an operator that is applied to a variable, constant, function or parenthetic expression that immediately follows it. The plus sign (+) and minus sign (−) can be used as prefix operators. The word NOT and its equivalent symbols are also prefix operators.

An infix operator applies to the operands on each side of an expression (for example 12<10).

Note: When used to perform arithmetic operations, the plus and minus signs are infix operators.

Arithmetic Operators
Arithmetic operators indicate that an arithmetic calculation is performed, as shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Precedence</th>
<th>Definition</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>80</td>
<td>Exponentiation</td>
<td>x**2</td>
<td>Raise x to the second power</td>
</tr>
<tr>
<td>*</td>
<td>60</td>
<td>Multiplication</td>
<td>6.5*salary</td>
<td>Multiply 6.5 by the value of salary.</td>
</tr>
<tr>
<td>/</td>
<td>60</td>
<td>Division</td>
<td>var/2</td>
<td>Divide the value of var by 2.</td>
</tr>
<tr>
<td>+</td>
<td>40</td>
<td>Addition</td>
<td>num+4</td>
<td>Add 4 to the value of num.</td>
</tr>
<tr>
<td>—</td>
<td>40</td>
<td>Subtraction</td>
<td>sale-discount</td>
<td>Subtract the value of discount from the value of sale.</td>
</tr>
</tbody>
</table>

The asterik (*) 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. If the comparison is true, the result is 1. If the comparison is false, the result is 0.

Comparison operators can be expressed as symbols or with their mnemonic equivalents, which are shown in the following table:
Table 1.2  Comparison Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Precedence</th>
<th>Mnemonic Equivalent</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>30</td>
<td>EQ</td>
<td>Equal to</td>
<td>x=5</td>
</tr>
<tr>
<td>=:</td>
<td>30</td>
<td></td>
<td>Equal to with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>30</td>
<td>EQ</td>
<td>Equal to</td>
<td>b==10</td>
</tr>
<tr>
<td>==:</td>
<td>30</td>
<td></td>
<td>Equal to with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>30</td>
<td></td>
<td>Not equal to</td>
<td>y!=3</td>
</tr>
<tr>
<td>!=:</td>
<td>30</td>
<td></td>
<td>Not equal to with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>30</td>
<td>LTE</td>
<td>Less than or equal to</td>
<td>hours&lt;=60</td>
</tr>
<tr>
<td>&lt;=:</td>
<td>30</td>
<td></td>
<td>Less than or equal to with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>30</td>
<td>GTE</td>
<td>Greater than or equal to</td>
<td>hours&gt;=20</td>
</tr>
<tr>
<td>&gt;=:</td>
<td>30</td>
<td></td>
<td>Greater than or equal to with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>30</td>
<td>LT</td>
<td>Less than</td>
<td>rate&lt;40</td>
</tr>
<tr>
<td>&lt;:</td>
<td>30</td>
<td></td>
<td>Less than with truncation to smaller length</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>30</td>
<td>GT</td>
<td>Greater than</td>
<td>rate&gt;10</td>
</tr>
<tr>
<td>&gt;:</td>
<td>30</td>
<td></td>
<td>Greater than with truncation to smaller length</td>
<td></td>
</tr>
</tbody>
</table>

**Numeric Comparisons**

Numeric comparisons in SAS are based on values. In the expression \( x \geq z \), if \( x \) has the value of 22 and \( z \) has the value of 7, then \( x \geq z \) has the value 1, or true. If \( x \) is 10 and \( z \) is 18, then the expression has the value 0, or false. If \( x \) and \( z \) each have the value 30, then the expression is true and has the value of 1.

Comparison operators often appear in IF-THEN statements. For example:

```plaintext
if sale > invoice then profit=2;
else profit=10;
```
A missing numeric value is smaller than any other numeric value, and missing numeric values have their own sort order.

**Character Comparisons**

You can perform comparisons on character operands, but the comparison always yields a numeric result (1 or 0). 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, \texttt{R} is greater than \texttt{G}. Therefore, this expression is true:

\texttt{'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 \texttt{m}, this expression is true:

\texttt{'C.Mills' < 'Charles Mills'}

Since trailing blanks are ignored in comparison, \texttt{dog} is equalivalent to \texttt{dog}. However, because blanks at the beginning and in the middle of a character value are significant to SAS, \texttt{dog} is not equivalent to \texttt{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:

\texttt{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:

- \texttt{if lastname>='F';}
- \texttt{if lastname>=:'F';}

**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.

**Logical (Boolean) Operators and Expressions**

Logical operators, also called Boolean operators, are usually used in expressions to link sequences of comparisons. The logical operators are shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Precedence</th>
<th>Mnemonic Equivalent</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>25</td>
<td>AND</td>
<td>Logical AND</td>
<td>((a&gt;b &amp; c&gt;d))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>Logical OR</td>
<td>((a&gt;b \text{ or } c&gt;d))</td>
</tr>
<tr>
<td>!</td>
<td>50</td>
<td>NOT</td>
<td>Logical NOT</td>
<td>not ((a&gt;b))</td>
</tr>
</tbody>
</table>

Using: CAS Procedure 31
**The AND Operator**

If both of the quantities are linked by AND are true (1), then the result of the AND operation is 1. Otherwise, the result is 0. For example, in the following comparison:

\[ \text{MSRP} > \text{Sale} \land \text{MSRP} < 20000 \]

the result is true (has a value of 1) only when both \( \text{MSRP} > \text{Sale} \) and \( \text{MSRP} < 20000 \) are 1 (true): that is, when MSRP is greater than Sale and MSRP is positive.

Two comparisons with a common variable linked by AND can be condensed with an implied AND. For example, the following two subsetting IF statements produce the same result:

- \( \text{if } 18 \leq \text{age} \land \text{age} \leq 65; \)
- \( \text{if } 16 \leq \text{age} \leq 65; \)

**The IN Operator**

The IN operator, which is a comparison operator, searches for character and numeric values that are equal to one from a list of values. The list of values must be in parentheses, with each character value in quotation marks and separated by either a comma or blank.

For example, in your report you only want to print the average household income in New York (NY), California (CA), and Texas (TX). You can use the IN operator which can select any states from your list.

\[ \text{where state in ('NY', 'CA', 'TX');} \]

For numeric comparison, you can use shorthand notation to specify a range of sequential integers to search.

\[ x = a \in (1, 2, 3, 4, 5, 6, 7); \]
\[ x = a \in (1:7); \]

You can use multiple ranges in the same IN list, and you can use ranges with other constants in an IN list. The following example shows a range that is used with other constants to test if \( X \) is 1, 2, 3, or 7.

\[ \text{if } x \in (7,1:3); \]

You can also use the IN operator to search an array of numeric values. For example, the following code creates an array \( n \), defines a constant \( g \), and then uses the IN operator to search for \( g \) in array \( n \).

**Note:** The array initialization syntax of array \( a \{10\} \) \( (2*1:5) \) creates an array that contains the initial values of 1, 2, 3, 4, 5, 1, 2, 3, 4, 5.

```plaintext
proc cas;
data _null_;    
array a{10} {2*1:5};
x=99;
y = x in a;
put y=;
a{5}=99;
y = x in a;
put y=;
run;
```

```
y=0  
y=1
```
The **MIN and MAX Operators**
The MIN and MAX operators are used to find the minimum or maximum value of two quantities. Surround the operators with the two quantities whose minimum or maximum value you want to know. The MIN (<> operator returns the lower of the two values. The MAX (<> operator returns the higher of the two values. For example, if invoice<sale, then invoice<>sale returns the value of invoice.

*Note:* In a WHERE statement or clause, the <> operator is equivalent to NE.

The **NOT Operator**
The prefix operator NOT is also a logical operator. The result of putting NOT in front of a quantity whose value is 0 (false) is 1 (true). That is, the result of negating a false statement is 1 (true). For example, if X=Y is 0 (false) then NOT(X=Y) is 1 (true). The result of NOT in front of a quantity whose value is missing is also 1 (true). The result of NOT in front of a quantity with a nonzero, nonmissing value is 0 (false). That is, the result of negating a true statement is 0 (false).

For example, the following two expressions are equivalent:

- `not (name='JONES')`
- `name ne 'JONES'`

Furthermore, NOT (A&B) is equivalent to NOT A|NOT B, and NOT (A|B) is the same as NOT A & NOT B. For example, the following two expressions are equivalent:

- `not (a=b & c>d)`
- `a ne b | c le d`

The **OR Operator**
If either of the quantities linked by an OR is 1 (true), then the result of the OR operation is 1 (true). Otherwise, the OR operation produces a 0. For example, consider the following comparison:

\[ a<b | c>0 \]

The result is true (with a value of 1) when A<B is 1 (true), regardless of the values of A and B. Therefore, it is true when either or both of those relationships hold.

### Missing Values

**Definition**

A *missing value* is a value that indicates that no data value is stored for the variable in the current observation. There are three types of missing values:

- numeric
- character
- special numeric

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

**Numeric Variables**

Within SAS, a missing value for a numeric variable is smaller than all numbers. If you sort your data set by a numeric variable, observations with missing values for that
variable appear first in the sorted data set. For numeric variables, you can compare special missing values with numbers and with each other. The following table shows the sorting order of numeric values:

<table>
<thead>
<tr>
<th>Sort Order</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallest</td>
<td>_</td>
<td>underscore</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td></td>
<td>.A-Z</td>
<td>special missing value A (smallest) through Z (largest)</td>
</tr>
<tr>
<td></td>
<td>-n</td>
<td>negative numbers</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>largest</td>
<td>+n</td>
<td>positive numbers</td>
</tr>
</tbody>
</table>

**Character Variables**

Missing values of character variables are smaller than any printable character value. Therefore, when you sort a data set by a character variable, observations with missing (blank) values for the BY variable always appear before observations in which values for the BY variable contain only printable characters. However, some usually unprintable characters (for example, machine carriage-control characters and real or binary numeric data that have been read in error as character data) have values less than the blank. Therefore, when your data includes unprintable characters, missing values might not appear first in a sorted data set.

**WHERE Expression Processing**

**Definition**

A WHERE expression defines a condition for selecting observations. A WHERE expression can be a variable name or constant, or it can be a SAS function with a sequence of operands and operators.

WHERE operand <operator>;

**operand**

something to be operated on. An operand can be a variable, a SAS function, or a constant.

**operator**

a symbol that requests a comparison, logical operation, or arithmetic calculation. All SAS expression operators are valid for a WHERE expression. In addition, you can use special WHERE expression operators that include BETWEEN, CONTAINS, LIKE, and IN.
When to Use a WHERE Expression

WHERE expression processing enables you to conditionally select a subset of observations, so that SAS processes only the observations that meet a set of specified conditions. For example, you have a SAS data set that contains employee salaries for 2014 and you only want to print those salaries greater than $60,000, but less than $100,000.

The BETWEEN — AND Operator

The BETWEEN-AND operator is a fully bounded range condition that selects observations in which the value of a variable falls within an inclusive range of values. You can specify the limits of the range as constants or expressions. Any range that you specify is an inclusive range, so that a value equal to one of the limits of the range is within the range.

The general syntax for using BETWEEN-AND is as follows:

```sql
WHERE variable BETWEEN value AND value;
```

For example, in your report you want only a list of employees whose taxes are between 0.30 and 0.50 you could use the following code to generate the list:

```sql
where taxes between salary*0.30 and salary*0.50;
```

Note: You can use comparison and numeric operators and the BETWEEN-AND operator together in the WHERE expression.

The LIKE Operator

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:

percent sign (%)
- specifies that any number of characters can occupy that position. The following WHERE expression selects all employees with a name that starts with the letter B. The names can be of any length.

```sql
where lastname like 'B%';
```

underscore (_)
- matches just one character in the value for each underscore character. You can specify more than one consecutive underscore character in a pattern, and you can specify a percent sign and an underscore in the same pattern.

You can use a SAS character expression to specify a pattern, but you cannot use a SAS character expression that uses a SAS function.

The CONTAINS Operator

The most common usage of the CONTAINS (?) operator is to select observations by searching for a specified set of characters within the values of a character variable. The position of the string within the variable's values does not matter. However, the operator is case sensitive when making comparisons.

Examples

You can use WHERE expression processing to subset a table by listing the rows and columns that you want to keep for your new table. You can also add a computed column and print to the log.

```sql
proc cas;
  output log;
```
columns=\{"x", "y", "z"\};
coltypes=\{"integers", "double", "string"\};
table = newtable("trouble", 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;

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7.8</td>
<td>abc34</td>
<td>0.51</td>
</tr>
<tr>
<td>4</td>
<td>10.4</td>
<td>abc44</td>
<td>0.38</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>abc54</td>
<td>0.31</td>
</tr>
<tr>
<td>5</td>
<td>7.8</td>
<td>abc35</td>
<td>0.64</td>
</tr>
<tr>
<td>5</td>
<td>10.4</td>
<td>abc45</td>
<td>0.48</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>abc55</td>
<td>0.38</td>
</tr>
</tbody>
</table>

**Functions**

**Overview**
The CASL language supports function calls within expression evaluation. A function can take multiple arguments. The function can return an error if the correct number of arguments were not provided. Variables that were created during the execution of a CASL function are local to that function. The CASL interface provides you with run time support to create and manage values that the routine may return. Functions can be defined in many ways:

- CASL supports internal functions that provides run-time support for your CASL program.
- The FUNCTION statement (FNC) can be used to defined functions using the CASL syntax that accepts a parameter and returns a value.
- PROC CAS may install functions to support native operations.
- CASL provides a default set of functions for regular operations. You can replace these functions with your own.
- DS2 functions are supported and you can use the FNC statement to get a list and description of the functions.

**FNC Statement**
The FNC statement is used to list all available functions by name and category with simple descriptions.
The following code displays all available functions within the FNC statement.

```plaintext
1? proc cas;
2? fnc cate;
3? run;
```

This is the output of all available functions.

```plaintext
array
bitwise
char
combinatorial
datetime
distance
financial
math
probability
quantile
random
special
statistics
trig
truncation
```

```plaintext
4? fnc bitwise; run;
```

```plaintext
band          (bitwise)    Returns the bitwise logical AND of two arguments.
blshift       (bitwise)    Returns the bitwise logical left shift of two arguments.
bnot          (bitwise)    Returns the bitwise logical NOT of an argument.
bor           (bitwise)    Returns the bitwise logical OR of an argument.
brshift       (bitwise)    Returns the bitwise logical right shift of two arguments.
bxor          (bitwise)    Returns the bitwise logical EXCLUSIVE OR of two arguments.
```

```plaintext
5? fnc bnot;
6? run;
```

```plaintext
bnot          (bitwise)    Returns the bitwise logical NOT of an argument.
double  bnot(double);
```

```plaintext
8? fnc "interest";
9? run;
```
Internal functions are called upon and cannot be replaced by function definitions.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>Determines the dimensions of a variable.</td>
<td>x[10] = 1; y = dim(x);</td>
</tr>
<tr>
<td>newtable</td>
<td>Creates a new table.</td>
<td>columns = {&quot;make&quot;, &quot;MSRP&quot;, &quot;HP&quot;, &quot;cylinders&quot;};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>coltypes ={&quot;varchar&quot;, &quot;int64&quot;, &quot;integer&quot;,&quot;int64&quot;};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>row1 = {&quot;dodge&quot;, 20000, 250, 4};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>row2 = {&quot;ford&quot;, 30000, 200, 6};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>table = newtable( &quot;cars&quot;, columns, coltypes, row1, row2);</td>
</tr>
<tr>
<td>add_table_attr</td>
<td>Adds attributes to a table.</td>
<td>rc = add_table_attr(table, &quot;Actionset&quot;, &quot;imstat&quot;, &quot;Action&quot;, &quot;Summary&quot;);</td>
</tr>
<tr>
<td>discard</td>
<td>Discards the variables that are listed as arguments.</td>
<td>x[10] = 1; y = 2; discard(x,y);</td>
</tr>
<tr>
<td>tabcolumns</td>
<td>Gets the types for a table.</td>
<td>coltypes = tabtypes(table);</td>
</tr>
</tbody>
</table>
### Function Name
### Definition
### Example

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>printtable</td>
<td>Prints table to a specified location.</td>
<td><code>printtable (table, &quot;&lt;fileref&gt;&quot;, columns);</code></td>
</tr>
<tr>
<td>addrow</td>
<td>Adds row to a table.</td>
<td><code>addrow(table, row1);</code></td>
</tr>
<tr>
<td>findtable</td>
<td>Searches a given value in the table.</td>
<td><code>table = findtable(result);</code></td>
</tr>
<tr>
<td>traceback</td>
<td>Returns a string consisting of traceback from the current function.</td>
<td><code>tb=traceback();</code></td>
</tr>
<tr>
<td>dictionary function</td>
<td>Returns the value of a key from a dictionary.</td>
<td><code>y=dictionary(dict, &quot;name&quot;);</code></td>
</tr>
<tr>
<td></td>
<td>Determines if the key exists in a dictionary.</td>
<td><code>y=exists(dict,&quot;name&quot;);</code></td>
</tr>
</tbody>
</table>

### Math Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>Minimum of a list of values</td>
<td><code>y = min(a, b, c, d, e);</code></td>
</tr>
<tr>
<td>max</td>
<td>Maximum of a list of values</td>
<td><code>y = max(a, b, c, d, e);</code></td>
</tr>
<tr>
<td>sqrt</td>
<td>Square root of a value</td>
<td><code>y=sqrt(z);</code></td>
</tr>
<tr>
<td>lgamma</td>
<td>Natural logarithm of a gamma function</td>
<td><code>y = lgamma(x);</code></td>
</tr>
<tr>
<td>exp</td>
<td>Value of e (the base of natural logarithms)</td>
<td><code>y = exp(x);</code></td>
</tr>
<tr>
<td>log</td>
<td>Function that returns the natural logarithm of x</td>
<td><code>y = log(x);</code></td>
</tr>
<tr>
<td>log2</td>
<td>Base 2 logarithm</td>
<td><code>y = log2(x);</code></td>
</tr>
</tbody>
</table>

### Exceptions

You can define exception-handler functions. All of the exception-handler functions take exactly one parameter which is a dictionary of information. Within the dictionary the context of the exception and parameters are specified.

This is an example of a handler for a system exception such as a floating-point error.

```plaintext
function myfphandler(env) do;
  put "error detected, using_MISSING_instead";
```
setexceptvalue("_MISSING_ ");
resume;
end do;
on_fpexception call fphandler;

The example above is telling the handler when you find an error use "_MISSING_". If SAS finds an error or a floating-point error in your data then it will display MISSING every time there is an error; otherwise, it will continue with the value that it is supposed to display.

Results: CAS Procedure

Result Table

The CAS result table is a table that is created as the result of an action. In addition to rows and columns, the table also contains labels and variable types. The table is the primary means to return information to CASL.

CASL offers a variety of operations on the CAS result table:

- You can reference particular row and column values within a result table.
- You can extract a row, column names, and types into a dictionary.
- You can subset a table by listing the rows and columns to be kept in the new table.
- You can use WHERE expression processing to create a new table with rows that match the WHERE expression.
- You can use the COMPUTE clause to create computed columns or to create an array with computed values from each row.
- You can iterate through the result that you receive from the submission of an action.

CASL enables you to create your own result table from the existing result table or as a subset or combination of other result tables.

Examples: CAS Procedure

Example 1: Set-up Program for PROC CAS

The following example shows you how to access your data, load the data into CAS, and perform any additional setup that is needed to run CASL statements. This example assumes that you have operating system level access to the server. You must be able to save the input data files in the path-to-data-directory. For more information on accessing a client-side data, see “Load a Client-Side File” in SAS Cloud Analytic Services: Accessing and Manipulating Data

Program

```r
options cashost="cloud.example.com" casport=5570; /*"*/
Example 2: Run an Action

This example runs the specified action, in this case, the listNodes action and displays the contents of the table to the Output Delivery System (ODS). For information about how to download and access the data, see Set Up Program for PROC CAS on page 40.

Program

```sas
proc cas; /*1*/
    action listnodes results=res submit; /*2 */
    put res; /*3 */
run;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The ACTION statement runs the specified action, in this case, the listNodes action. The results of the listNodes action are stored in the variable named Res. The SUBMIT option specifies to run the action synchronously as programming statements are evaluated. The listNodes action does not accept any parameters, so no parameters are specified.

3 The PUT statement displays the contents of the variable Res. The listNodes action returns a table and the PUT statement supplies the table to the Output Delivery System (ODS) for display.
The following output shows sample results for eight machines.

**Output 1.2**  Output: HTML Output for listNodes action

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Connected</th>
<th>Server Port</th>
<th>Primary IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdcgrd001</td>
<td>controller</td>
<td>1</td>
<td>58475</td>
<td>10.124.3.9</td>
</tr>
<tr>
<td>rdcgrd002</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd003</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd004</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd005</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd006</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd007</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rdcgrd008</td>
<td>worker</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Example 3: Define a New Function**

This example defines a new function and executes the function using the IF-ELSE/THEN, DO, and PRINT statements. This example also uses an internal function, `factorial`, to define and execute the newly created function `x`.

**Program**

```sas
proc cas; /*#1*/
  function factorial(x); /*#2*/
    if (x < 1.0) then return(x); /*#3*/
    else do;
    return exp(lgamma (x+1));
    end; /*#4*/
  end;

  function factorial(x);
    if (x <= 2.0) then return(x);
    else return( factorialr(x-1)*x);
  end;
run;

do i = 1 to 9; /*#5*/
x = factorial(i);
  print "Factorial (" i ") = " put(x,best9.);/*#6*/
end;
x = factorialr(75);
  print *Factorial of 75 = " x;
run;
x = factorialr(75.0);
  print *Factorial of 75.0 = " x;
run;
```

1. The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.
2. The FUNCTION statement creates a new function that takes one argument.
This function uses the `lgamma` and `exp` internal functions to calculate the factorial of a number. The IF-THEN/ELSE statement asks the function `x` to find values where the value of `x` is less than 1.0, if it is then return the value of `x`. The ELSE statement asks to return the run-time math expression function `lgamma` to be multiplied by the value of `x` and added 1.

The END statement ends the function processing.

The DO statement iterates over the list from 1 to 9.

The PRINT statement prints the results. Put acts as a DATA step.

### Log 1.1  Output Log: Define A New Function

| Factorial (1) | 1 |
| Factorial (2) | 2 |
| Factorial (3) | 6 |
| Factorial (4) | 24 |
| Factorial (5) | 120 |
| Factorial (6) | 720 |
| Factorial (7) | 5040 |
| Factorial (8) | 40320 |
| Factorial (9) | 362880 |
| Factorial of 75 | 0 |

Factorial of 75.0 = 2.480914E109

---

### Example 4: Use Run-time Math Functions

**Program**

In this example we not only create a new function but also use built-in run-time math functions to calculate the probability of the number of people who share the same birthday in the same room. This example uses built-in internal functions `lgamma` and `exp` to calculate the factorial of a number.

```
proc cas; /* 11 */
  function SharedFeature (feature,number) /* 12 */
    if p = exp(lgamma (feature+1)- lgamma(feature-number+1) - number*log(feature)); /* 13 */
      else return (1-p); /* 14 */
  end; /* 15 */
  do n over {3 10 22 23 50 75}; /* 16 */
    p = SharedFeature(365,n) /* 17 */
    print "Chance at least 2 out of " put(n,best3.) /* 18 */
    " share same birthday = " put(p,best8.4) /* 19 */
  end;
```

1. The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2. The FUNCTION statement creates a new function. This function contains two arguments and calculates the probability that at least 2 items share the same feature given `number` of items and `features`.

---

Example 4: Use Run-time Math Functions
3 The IF-THEN/ELSE statement executes statements that meet specific conditions. This function uses the `lgamma` and `exp` internal functions to calculate the factorial of a number.

4 ELSE returns the value of one subtracted from the value of $p$.

5 The END statement ends the function processing.

6 The DO statement iterates over a list.

7 The variable $p$ is defined here. Use the function `SharedFeature` to calculate the probability that at least two people have the same birthday given the number of people in the room.

8 The PRINT statement prints the results. Put acts as a DATA step.

   Note: Functions can be recursive and they can execute any CASL statement. Functions can also be re-defined.

Log 1.2  Output: Log: Probability of Two People Sharing the Same Birthday in the Same Room

<table>
<thead>
<tr>
<th>Chance at least 2 out of</th>
<th>Share same birthday</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>0.0082</td>
</tr>
<tr>
<td>10</td>
<td>22</td>
<td>0.4757</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>0.5073</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>0.9704</td>
</tr>
<tr>
<td>75</td>
<td>75</td>
<td>0.9997</td>
</tr>
</tbody>
</table>

Example 5: Train a Decision Tree Model and Score Data

This example shows you how to train a decision tree model and score data generated by CASL actions.

Program

```
cas casauto;
libname mycas cas sessref=casauto;
data mycas.golf; /*```
  format outlook $8.;
  format temperature best10.;
  format humidity best10.;
  format windy $5.;
  format golf $10.;
  input outlook $ 1-8 temperature humidity windy $ 16 - 21 golf $22 - 32;
datalines;
sunny 85 85 false Don't Play
sunny 80 90 true  Don't Play
overcast 83 78 false Play
rain 70 96 false Play
rain 68 80 false Play
rain 65 70 true  Don't Play
overcast 64 65 true  Play
sunny 72 95 false Don't Play
sunny 69 70 false Play
rain 75 80 false Play
```
sunny    75  70 true Play
overcast  72  90 true Play
overcast  81  75 false Play
rain      71  80 true Don't Play
;
run;
filename score *~/score_golf.sas*; /*#2*/
proc cas;
    decisionTree.dtreeTrain result=r / /*#3*/
        table={name = "golf"}
        inputs={"outlook", "windy", "humidity", "temperature"}
        target="golf"
        maxlevel =4
        maxbranch=2
        nbins =5
        binorder =1
        varImp =true
        code={labelid=999, comment=true, tabForm=true};
run;
    print r['ModelInfo'];
    print r['DTreeVarImpInfo'];
run;
    saveresult r['CodeGen'] file=score; /*#4*/
run;
quit;
data mycas.more_golf; /*#5*/
    format outlook     $8.;
    format temperature best10.;
    format humidity    best10.;
    format windy       $5.;
    *format golf       $10.;
    input outlook $ 1-8 temperature humidity windy $ 16 - 21 /* golf $22 - 32 */;
datalines;
sunny    75 85 true
overcast  83  78 false
sunny    68 80 false
;
run;
proc cas;
    code = readfile("score"); /*#6*/
dcode =
    "data more_golf_scored;
    set more_golf;
    || code ||
    "run;";
run;
dataStep.runCode / code = dcode; /*#7*/
run;
table.fetch / table = "more_golf_scored"; /*#8*/
run;
quit;

libname casuser v9 '/u/casuser;
data casuser.golf;
    set mycas.golf;
run;
data casuser.more_golf;
  set mycas.more_golf;
run;

1 Generate, save, and run DATA step score code using CASL. The DATA step in CAS operates on CAS tables. Input and output data sets must use the CAS LIBNAME engine. The engine enables the DATA step to fetch column metadata for CAS tables when compiling the program. Data sets that use other engines must be loaded into CAS, or a caslib must be defined for that data source.

2 The FILENAME statement associates a SAS fileref with an external file. In this example, the FILENAME statement associates score_golf.sas with the score file which is external.

3 The decision tree action set provides actions that can generate DATA step scoring code for decision tree models. The dtreeTrain action trains a decision tree. For more information on the syntax of dtreeTrain, see “Train decision tree” in SAS Visual Analytics Programming Guide.

4 The SAVERESULT statement saves the DATA step score code to a SAS data set.

5 DATA step creates new data to score. Notice the commented out code in the DATA step, golf has been left out of the DATA step.

6 Use Readfile to read the contents of the file. The Readfile function reads the contents of the file given into the variable as a string.

7 The dataStep.runCode action is scoring new observations. The scored data includes a prediction for playing golf and a predicted probability.

8 Fetch rows from a table. Use the format parameter to apply formats to the variables. Use sortBy to specify the variables and variable settings for sorting results. Use the parameter to specify the ordinal position of the last row to return.
**Output 1.3**  HTML Output: Trained Decision Tree

```
Decision Tree for GOLF

<table>
<thead>
<tr>
<th>Decision Tree for GOLF</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Tree Nodes</td>
<td>3.000000</td>
</tr>
<tr>
<td>Max Number of Branches</td>
<td>2.000000</td>
</tr>
<tr>
<td>Number of Levels</td>
<td>2.000000</td>
</tr>
<tr>
<td>Number of Leaves</td>
<td>2.000000</td>
</tr>
<tr>
<td>Number of Bins</td>
<td>5.000000</td>
</tr>
<tr>
<td>Minimum Size of Leaves</td>
<td>5.000000</td>
</tr>
<tr>
<td>Maximum Size of Leaves</td>
<td>9.000000</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>4.000000</td>
</tr>
<tr>
<td>Confidence Level for Pruning</td>
<td>0.250000</td>
</tr>
<tr>
<td>Number of Observations Used</td>
<td>14.000000</td>
</tr>
<tr>
<td>Misclassification Error (%)</td>
<td>28.571429</td>
</tr>
</tbody>
</table>
```

**Output 1.4**  HTML Output: New Data Set

```
<table>
<thead>
<tr>
<th>CAS Library</th>
<th>Name</th>
<th>Number of Rows</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASUSER(casuser)</td>
<td>more_golf</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>CAS Library</th>
<th>Name</th>
<th>Number of Rows</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASUSER(casuser)</td>
<td>more_golf_scored</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Example 6: Computing Partial Dependency

This example uses decision tree to train a regression model. The example takes one variable and integrates other variables by creating a fake data set (_partial_data) and uses the datastep action and dtreescore to score the data set and run a summary to get the predicted response.

Program

```sas
proc cas ;                          /* 1 */
file log;                            /* 2 */
datastep.runcode/ code='             /* 3 */
data simuData;
do i=1 to 100;                          /* 4 */
x1= rand("uniform");
x2= rand("uniform");
x3= int(4*rand("uniform"));
p= 1/(1+exp(-(sin(3*x1)-4*x2+x3)));
y=x1*x2+x3;
output;
end;                                    /* 5 */
drop i;
run;
';
run;
quit;
```

1. The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2. The FILE statement enables you to specify a different location on output. In this example, the FILE statement specifies the log as the location on output.

3. The datastep.runCode action runs a DATA step program to this action as a string.

4. The DO statement iterates over the list. In this example, the DO statement identifies the loop variable as `i` and loops from 1 to 100.

5. The END statement ends the DO processing.
**Log 1.3  Output: Log**

NOTE: Active Session now CASAUTO.

<table>
<thead>
<tr>
<th>CAS Library</th>
<th>Name</th>
<th>Number of Rows</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASUSERHDFS(sasdemo)</td>
<td>simuData</td>
<td>444800</td>
<td>5</td>
</tr>
</tbody>
</table>

/* Train a Simple Regression Tree*/
proc cas ;
decisionTree.dtreetrain result=r /table={name='simuData'} /*
#1
*/
  inputs=${x1-x3}
  target='y' casout={name='mytree',
                    replace=1}
  nbins=50 maxbranch=2 maxlevel=5 binorder=true
  mergebin=true leafsize=1;
print r; /*#2*/
run;
quit;

1  The action set decisionTree calls on the action dtreetrain to train a decision tree for classifications or regressions.

2  The PRINT statement prints the results.

proc cas ;
/* Compute the partial dependency for one point.*/
function partialDependencyForOnePoint(inputTableName, modelName, /*
#1*/
  partialVarName, colName, debug);
  compExpr=partialVarName || '=' || colName || ';'; /*#2*/
  if (debug=1) then print compExpr; /*#3*/
decisionTree.dtreescore result=r/table={name=inputTableName,
  compvars=partialVarName, comppgm=compExpr} /* recreate original x */
model={name=modelName}
  casout={name='_scored_out', replace=1}
  path=0;
  if (debug=1) then print r;
/* get the predicted y */
simple.summary result=r/ table={name='_scored_out', vars='_DT_PredMean_'}; /*#4*/
/* drop this table */
  if (debug=0) then do;
    droptable/table='_scored_out';
  end;
return r['summary'][1, 'mean'];
end func;

/* inputTableName: input table */
/* modelName: model name for dtreecore */
/* partialVarName: the variable name to be computed for dependency */
/* inputVarNameList: the variables need be integrated out */
/* lowerBound: lower bound of partialVarName */
/* upperrBound: upper bound of partialVarName */
/* nPoints: number of points */
/* debug: if 1, enable debug mode */

Example 6: Computing Partial Dependency
function partialDependency(inputTableName, modelName, partialVarName, inputVarNameList, lowerBound, upperBound, nPoints, debug);

/* generate a new set of column names */
newColumns={};
do i=1 to nPoints;
   newCol=partialVarName || '_' ||(string)i;
   newColumns=newColumns + $(newCol);
end;
codeExpr="data _partial_data;"
     || "set " || inputTableName || ";"
     || "keep ";

nInputVars=dim(inputVarNameList);
do i=1 to nInputVars;
   codeExpr=codeExpr || inputVarNameList[i] || ' ';
end;
do i=1 to nPoints;
   codeExpr=codeExpr || newColumns[i] || ' ';
end;
codeExpr=codeExpr || ';
myx=lowerBound;
width=(upperBound-lowerBound)/nPoints;
do i=1 to nPoints;
   codeExpr=codeExpr || newColumns[i] || '=' || (string) (myx+(i-1)*width) || ' ';
end;
codeExpr=codeExpr || 'output;';
if (debug=1) then print codeExpr;
datastep.runcode result=r / code=codeExpr;

/* now we get a wide table and then run each scoring action for each selected column */
myx=lowerBound;
do i=1 to nPoints;
   x=myx+(i-1)*width;
   y=partialDependencyForOnePoint('_partial_data', modelName, partialVarName, newColumns[i], debug);
   res.x[i]=x;
   res.y[i]=y;
end;

/* drop the temp table */
if (debug=0) then do;
   droptable/table='_partial_data';
end;
return res;
end func;
run;

/* let us try it by integrating x2 and x3 out */
otherVars={'x2', 'x3'};
res=partialDependency('simuData', 'mytree', 'x1', otherVars, 0, 1, 10, 0);

/* print result into the log */
print res;
run;
1. The FUNCTION statement creates a new function named partialDependencyForOnePoint and has five arguments named inputTableName, modelName, partialVarName, colName, and debug.

2. A new variable is created named `compExpr`. This variable is created using the Assignment statement. For more information see “ASSIGNMENT Statement” on page 8.

3. The IF statement executes statements that meet specific conditions. In this example, if the argument debug equals to 1 then using the PRINT the value of the variable `compExpr`.

4. The action set simple executes the action summary to generate descriptive statistics about the data and the results are saved into a new result variable named r.

```
Log 1.4  Output: Log: Output from Variable Res

{x={0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9},y={1.6166262987,1.6166262987,1.6166262987,1.6480508194,1.6480508194,1.6635301729,1.8676608205,1.8676608205,1.8676608205}}
```

```
Output 1.6  Results: Computing Partial Dependency

r: Results from decisionTree.dtreetrain

<table>
<thead>
<tr>
<th>Decision Tree for SIMUDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Tree Nodes</td>
</tr>
<tr>
<td>Max Number of Branches</td>
</tr>
<tr>
<td>Number of Levels</td>
</tr>
<tr>
<td>Number of Leaves</td>
</tr>
<tr>
<td>Number of Bins</td>
</tr>
<tr>
<td>Minimum Size of Leaves</td>
</tr>
<tr>
<td>Maximum Size of Leaves</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Alpha for Cost-Complexity Pruning</td>
</tr>
<tr>
<td>Number of Observations Used</td>
</tr>
<tr>
<td>Maximum STD of Leaves</td>
</tr>
<tr>
<td>Minimum STD of Leaves</td>
</tr>
<tr>
<td>Mean Squared Error</td>
</tr>
</tbody>
</table>
```

```
Output CAS Tables

<table>
<thead>
<tr>
<th>CAS Library</th>
<th>Name</th>
<th>Number of Rows</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASUSERHDFS(casuser)</td>
<td>mytree</td>
<td>31</td>
<td>23</td>
</tr>
</tbody>
</table>
```
Example 7: Subsetting A Computed Column

This example uses CAS actions to retrieve rows from a SAS data set, computes a ratio for all rows, subsets the results, and saves the table to disk. The example uses CAS statements, actions, and the CASUTIL procedure.

Program

```
proc casutil casout="hps"; #1
  load data=sashelp.cars;
proc cas;
  table.recordCount result=count / #2
    table="cars";
run;
  table.fetch result=fetchr; #3
    table="cars"
    to=findtable(count)[1,1];
run;
lowMSRP=sort(findtable(fetchr), "MSRP")[1:5,"MODEL","MAKE","MSRP"]; #4
  print lowMSRP;
run;
computedcolumn=findtable(fetchr). #5
  compute({"ratio","Invoice/MSRP",best5.3},Invoice/MSRP)
    [,{"MODEL", "MAKE","MSRP","ratio"}];
subset= sort(computedcolumn,"ratio")[1:5]; #6
  print subset;
run;
nrows=findtable(count)[1,1];
  print "total rows:" nrows;
run;
restable=sort(findtable(fetchr,"MSRP")[nrows-5:nrows,"model","make","msrp"]); #7
  saveresult lowMSRP dataout=sasuser.lowMSRP;
  saveresult subset dataout=sasuer.subset;
run;
print restable;
run;
```

1. The CASUTIL procedure loads a SAS data set.
2. The recordCount action shows the number of rows in the CARS table and saves the record count in the variable `count`.
3. The fetch action fetches rows from the table CARS.
4. The assignment statement defines the variable `lowMSRP`. The variable `lowMSRP` contains five vehicles with the lowest MSRP.
5. The assignment statement defines the variable `computedcolumn`. The `compute` function calculates an invoice to MSRP ratio for all rows.
6. The assignment statement defines the variable `subset`. The `sort` function sorts and subsets the five lowest ratios. The total number of rows is displayed in the SAS log.
7. The SAVERESULT statement saves the CAS tables to disk. The output tables are saved as `sas7bdat` files.
### Output 1.7  LowMSRP Result Table

#### LowMSRP: Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Make</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rio 4dr manual</td>
<td>Kia</td>
<td>$10,280</td>
</tr>
<tr>
<td>Accent 2dr hatch</td>
<td>Hyundai</td>
<td>$10,539</td>
</tr>
<tr>
<td>Echo 2dr manual</td>
<td>Toyota</td>
<td>$10,760</td>
</tr>
<tr>
<td>Ion1 4dr</td>
<td>Saturn</td>
<td>$10,995</td>
</tr>
<tr>
<td>Rio 4dr auto</td>
<td>Kia</td>
<td>$11,155</td>
</tr>
</tbody>
</table>

### Output 1.8  Subset Result Table

#### subset: Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Make</th>
<th>MSRP</th>
<th>Invoice/MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>911 Carrera 4S coupe 2dr (convert)</td>
<td>Porsche</td>
<td>$84,165</td>
<td>0.858</td>
</tr>
<tr>
<td>LX 470</td>
<td>Lexus</td>
<td>$64,800</td>
<td>0.871</td>
</tr>
<tr>
<td>SC 430 convertible 2dr</td>
<td>Lexus</td>
<td>$63,200</td>
<td>0.871</td>
</tr>
<tr>
<td>LS 430 4dr</td>
<td>Lexus</td>
<td>$55,750</td>
<td>0.871</td>
</tr>
<tr>
<td>GS 430 4dr</td>
<td>Lexus</td>
<td>$48,450</td>
<td>0.872</td>
</tr>
</tbody>
</table>
Recommended Reading

- *Getting Started with CASL*
- *SAS Cloud Analytic Services: Accessing and Manipulating Data*
- *SAS Viya: System Programming Guide*
- *SAS Visual Analytics Programming Guide*
- *SAS Visual Data Mining and Machine Learning: Programming Guide*

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

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: [sas.com/store/books](http://sas.com/store/books)
Gain Greater Insight into Your SAS® Software with SAS Books.

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

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