Creating a Rule That Uses Session Context to Qualify Customer

Teradata RTIM tracks session-specific information that is not saved after the session. You can use this information to create rules based on the length of time the customer has been in a session at the channel, whether the customer is in more than one channel, or the length of time since the last request.

1. Drag the **Contact Selection** function from the palette and drop it on the canvas.
2. In the edit panel, select the **CONTEXT** tab.

Note:
All of the **CONTEXT** fields are optional.

Setting	Value
Channel	A channel in the current session.
Interaction Point	An interaction point in the current session.
Multi-Channel Status	This setting applies only when the customer has an active session in more than one channel; for example, when a customer is logged in to the website and talking to someone at the call center at the same time.
Channel Interaction Length (Minutes)	Minimum session length in the specified channel.
Total Interaction Length (Minutes)	If the customer has an active session in only one channel, then the total interaction length is the same as the channel interaction length.
Time Since Last Request (Minutes)	Time since any request has been received from any channel in the current session.
New Channel Status	A Boolean setting that evaluates to true the first time the customer interacts with the channel during the current session.
New Interaction Point	A Boolean setting that evaluates to true the first time the customer interacts with the interaction point during the current session.

3. To test whether a customer began an active session in the Web channel at least 10 minutes ago and called technical support at least five minutes ago, specify the following:

Setting	Value
Channel	Call Center
Channel Interaction Length (Minutes)	5
Total Interaction Length (Minutes)	10

4. Click **APPLY**.

Creating Rules with Expressions

A rule expression is a rule that you type using the functions that Teradata RTIM supports. You can include rule expressions with other nodes in the rule criteria builder.

1. Drag the **Expression** function onto the canvas.
   The **Expression** editing panel is displayed.
2. Type the rule expression into the field, then click **APPLY**.
   For information about the available functions you can type, see Rule Expression Reference.

Rule Expression Use Cases

In some cases, you may need to specify complex rules that are easier to specify by typing them directly into a single field. Almost any rule is possible with rule expressions including rules that include arithmetic. The following use cases provide some examples.

Rule Expression Use Case 1

A store wants to qualify the Soft_drink_offer message for customers by defining a rule that uses several attributes from a single complex attribute category.

The complex attribute category named cart has three attributes: product, price, and discount. Each item in the customer's shopping cart can have values for all of these attributes.

For this example, the user's cart contains these values for the attributes:

- product: soda, potato chips, gum
- price: $1, $0.50, $0.50
- discount: $0.50, $0.25, $0

The Soda Offer rule qualifies the Soft_drink_offer message for the customer only when the following is true:

(product contains 'soda' AND the soda [price-discount] is greater than zero AND (the soda [price-discount] is greater than the 'potato chips' [price-discount]))

Sample Rule Expression Syntax for Use Case 1

The following shows the rule expression syntax for the Soda Offer rule for the Soft_drink_offer message.

Note:
The following example includes line numbers for reference that are not part of the rule expression. Do not include line numbers when you type your rule expressions.

```
001: set("cart", get("Account.cart")) &&
002: set("filterOnSoda", filter(cart, "product", "soda")) &&
003: count(filterOnSoda) > 0 &&
004: set("filterOnPotatoChip", filter(cart, "product", "potato chips")) &&
005: count(filterOnPotatoChip) > 0 &&
006: set("sodaItem", get(filterOnSoda, 0)) &&
007: set("potatoChipItem", get(filterOnPotatoChip, 0)) &&
008: set("sodaPriceDiscount", value(get(sodaItem, "price")) -
value(get(sodaItem, "discount"))) &&
009: set("potatoChipDiscount", value(get(potatoChipItem, "price")) -
value(get(potatoChipItem, "discount"))) &&
010: sodaPriceDiscount > potatoChipDiscount
```

Here is a brief explanation of what is happening in the lines of the rule expression:

1. Line 001 sets up the cart in a variable. The word Account refers to a complex attribute category; you must use the word Account rather than Complex.

2. Line 002 sets a filtered view of the cart in a variable.
3. Line 003 counts filtered soda to ensure there is at least one soda.
4. Lines 004 and 005 are similar to 002 and 003.
5. Lines 006 and 007 get the first item from the filtered views.
6. Lines 008 and 009 get the priceDiscount for the test.
7. Line 10 determines whether the soda discount is greater than the potato chip discount.

Rule Expression Use Case 2

For this use case, the rule qualifies the message named combo discount for customers by comparing attribute values from more than one complex attribute category.

The two complex attribute categories are cart and wallet.

These are the associated attribute values for the complex attribute category named cart:

price	upc
1.29	123
0.5	245

The following are the associated attribute values for the complex attribute category named wallet:

discount	upc
20	123
30	245

The rule qualifies the combo discount message for the customer only when the following is true:

The cart and wallet complex attribute categories both have upc attribute values of 123 and 245 AND the final price for upc 123 is greater than the final price for upc 245. Calculation of the final price of a the upc is made by price - (price * (discount / 100)).

Sample Rule Expression Syntax for Use Case 2

The following shows the rule expression syntax for the combo discount message.

```
set("cart", get("Account.cart")) &&
set("wallet", get("Account.wallet")) &&
set("filterCartOn123", filter(cart, "upc", "123")) &&
count(filterCartOn123) > 0 &&
set("filterWalletOn123", filter(wallet, "upc", "123")) &&
count(filterWalletOn123) > 0 &&
set("filterCartOn245", filter(cart, "upc", "245")) &&
count(filterCartOn245) > 0 &&
set("filterWalletOn245", filter(wallet, "upc", "245")) &&
count(filterWalletOn245) > 0 &&
set("cart123", get(filterCartOn123, 0)) &&
set("wallet123", get(filterWalletOn123, 0)) &&
set("price123", value(get(cart123, "price"))) &&
set("discount123", value(get(wallet123, "discountPercent"))) &&
set("adjustedPrice123", price123 - (price123 * (discount123 / 100))) &&
set("cart245", get(filterCartOn245, 0)) &&
set("wallet245", get(filterWalletOn245, 0)) &&
```

```
set("price245", value(get(cart245, "price"))) &&
set("discount245", value(get(wallet245, "discountPercent"))) &&
set("adjustedPrice245", price245 - (price245 * (discount245 / 100))) &&
adjustedPrice123 > adjustedPrice245
```

Rule Expression Use Case 3

In this use case, the following rule uses the customer's transaction profile to qualify a message.

These are the associated attribute values for the complex attribute category named transaction:

balance	date	name
10000	Oct 1, 2010	Airline Miles
50000	Oct 27, 2013	Hotel Points

The rule qualifies the offer_a message for the customer when the following is true:

The complex attribute category named transaction contains balance >= 50000 and the date associated with the transaction is within the last 30 days.

Sample Rule Expression Syntax for Use Case 3

The following shows rule expression syntax for qualifying a rule based on a customer's transaction activity.

```
set("txAccount", get("Account.transaction")) &&
set("txFilter", filter(txAccount, "name", "Airline Points")) &&
set("tx", get(txFilter, 0)) &&
set("txBalance", value(get(tx, "balance"))) &&
set("txDate", get(tx, "date")) &&
txBalance >= 50000 &&
time(parseDate(txDate, "MMM D, yyyy")) > time(now) - 1000 * 60 * 60 * 24 * 30
```

Rule Expression Reference

For convenience, all of the functions are described in alphabetical order, but the most powerful function and the one that you use most often when building expressions for rules is `get`. That section contains the bulk of information about rule expressions.

Note:
The rule expression functions are similar to JavaScript. Users who write rule expressions should have an understanding of JavaScript, the Java Pattern class, and regular expressions.

You can use the ! (not) operator in rule expressions. The following is an example:

```
!(get("Customer.general.height") == "short")
```

This could also be expressed as shown in the following example:

```
get("Customer.general.height") != "short"
```

The Java Expression Parser(JEP) math functions are fully supported in the rules engine. The following are the supported functions:

```
boolean ruleResult = getRulesResult("2 + 3 == 5");
assertTrue(ruleResult);
ruleResult = getRulesResult("2 * 3 == 6");
assertTrue(ruleResult);
ruleResult = getRulesResult("12 / 3 == 4");
assertTrue(ruleResult);
ruleResult = getRulesResult("12 - 3 == 9");
assertTrue(ruleResult);
ruleResult = getRulesResult("2 ^ 3 == 8");
assertTrue(ruleResult);
ruleResult = getRulesResult(" 1 - 3 == -2");
assertTrue(ruleResult);
ruleResult = getRulesResult(" 5 - 3 != 3");
assertTrue(ruleResult);
ruleResult = getRulesResult(" 1 - 3 == -2 && 5 - 3 != 3");
assertTrue(ruleResult);
ruleResult = getRulesResult(" 1 - 3 == 5 || 5 - 3 != 3");
assertTrue(ruleResult);
ruleResult = getRulesResult("! (2 == 3)");
assertTrue(ruleResult);
ruleResult = getRulesResult("sqrt(16)== 4");
assertTrue(ruleResult);
String expr = " set(\"attr\", 25) && (sqrt(get(\"attr\")) == 5) ";
ruleResult = getRulesResult(expr);
assertTrue(ruleResult);
expr = " set(\"attr\", 20) && (get(\"attr\") + get(\"attr\") == get(\"attr
\")*2) ";
ruleResult = getRulesResult(expr);
assertTrue(ruleResult);
ruleResult = getRulesResult("ln(1)== 0");
assertTrue(ruleResult);
ruleResult = getRulesResult("log(100)== 2");
assertTrue(ruleResult);
ruleResult = getRulesResult("mod(26,3) == 2");
assertTrue(ruleResult);
ruleResult = getRulesResult("26 % 3 == 2");
assertTrue(ruleResult);
ruleResult = getRulesResult("abs(-26) == 26");
assertTrue(ruleResult);
ruleResult = getRulesResult("rand() > 0");
assertTrue(ruleResult);
ruleResult = getRulesResult("sin(0) == 0");
assertTrue(ruleResult);
ruleResult = getRulesResult("cos(0) == 1");
assertTrue(ruleResult);
ruleResult = getRulesResult("sin(3.14/2) < 1 && sin(3.14/2) > 0.95");
assertTrue(ruleResult);
ruleResult = getRulesResult("cos(3.14/2) > 0 && cos(3.14/2) < 0.05");
assertTrue(ruleResult);
ruleResult = getRulesResult("sum([16,20,20]) == 56");
assertTrue(ruleResult);
```

avg

Returns the average of a set of values.

The following example finds the values of all price attributes in the complex attribute category named cart, adds the prices, divides the sum by the number of prices, and returns the average. The rest of the formula then takes that average and returns true if the average is greater than `1`.

```
avg(
    get("Account.cart"),
     "price"
) > 1
```

By default, the `avg` function accesses the most current data. The `Session` modifier enables the `avg` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute rather than a complex attribute with the `avg` function.

between

Returns all the values between the given parameter values including the start and end values. The `between` function is used only within the context of the `filter` or `is` functions.

```
count(
                filter(
                                get("Account.cart"),
                                "price",
                                between(40,45)
                )
) == 3
```

The example returns true if there are three rows for price that have a value greater than or equal to 40 and less than or equal to 45.

conditional

Controls the evaluation of other expressions in the entire expression tree. The `conditional` function is usually used as the first function in the expression stack.

```
conditional(
         get(”Account.dates.expiration”),
         NaN,
         false
) &&
set("fixedDateMillis",
    time(get(get("Account.dates.expiration "),0),"day")
) &&
fixedDateMillis > time("2099-01-01T00:00:00.000Z","day")
```

This expression evaluates to false if there is no tag history for expiration dates in the session. The evaluation of the expression stops after the `conditional` function because the tag value is null and the `get` function evaluates to not a number (NaN), which is the second parameter in the `conditional` function. The third

parameter is the value of the entire expression tree in this case. If there is tag history for expiration dates, the rest of the expression is evaluated.

contains

Returns all the values that are contained within another list of values. The `contains` function is used only within the context of the `filter` or `is` functions.

Example 1:

```
count(
                filter(
                                get("Account.cart"),
                                "items",
                                contains(list("3","4"))
                )
) >0
```

The example returns true when there are any rows for items that have the values 3 or 4.

Example 2:

```
count(
                filter(
                                get("Account.cart"),
                                "products",
                                contains(get("Account.clearance.products"))
                )
) >0
```

The example returns true when there are any rows for the cart products that are in the list of clearance products.

count

Returns a count of the list of data within a set.

In the following example, the expression gets the session data for the Customer.nav.page attribute and filters that session data for attribute values beginning with the word Credit. The expression counts the number of filtered attributes and returns true if the number is greater than or equal to `3`.

```
count(
      filter(
             get("Session.Customer.nav.page"),
             pattern("Credit.*")
      )
) >= 3
```

By default, the `count` function accesses the most current data. The `Session` modifier enables the `count` function to access the session data, which is *all* of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute with the `avg` function.

distinct

Returns a list of values that are unique within a set.

In the following rule expression, the `distinct` function returns the set of unique price attribute values from the complex attribute category named cart. The following expression evaluates as true when the count of unique price attribute values from the cart complex attribute category is greater than `13`.

```
count(
    distinct(
        get("Account.cart"),
         "price"
    )
) > 13
```

By default, the `distinct` function accesses the most current data. The `Session` modifier enables the `distinct` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute with the `distinct` function.

equalTo

Returns true for numerical attribute value equal to a given value. The `equalTo` function is used only within the context of the `filter` or `is` functions.

In the following rule expression, the `equalTo` function returns true when the complex attribute category named cart has two items where the value for the price attribute is equal to 1.25.

```
count(
    filter(
       get("Account.cart"),
       "price",
       equalTo(1.25)
    )
) == 2
```

filter

Returns a filtered list of values by testing a condition.

The following example determines whether the cart has three toy trucks. The rule expression gets the groups of attributes for the cart complex attribute category. Next, the rule expression filters the groups in the cart complex attribute category for all the groups that contain a product attribute with the value ToyTruck. The rule expression evaluates to true when the cart complex attribute category has three product attributes with the value ToyTruck.

```
count(
  filter(
     get("Account.cart")
      "product",
     "ToyTruck"
```

```
  )
) == 3
```

formatDate

Takes a datetime value expressed as time in milliseconds since 1970/1/1 00:00:00 GMT and returns it in the Java date-expression pattern specified.

The following expression returns true when the current date is January 17.

```
formatDate(now, "MM-dd") == "01-17"
```

formatDateTime

Formats each time-based value into the specified time interval. The `formatDateTime` function is used only within the context of a filter.

```
count(
       filter(
                get("Account.dates"),
                "expiration",
                formatDateTime("month"),
                equalTo(time(now,"month"))
       )
) > 3
```

This example returns true if there are more than three expiration dates this month.

Note:
See the time function for the supported time periods.

formatTimePeriod

Formats each time-based value into a numeric representation of the specified time interval. The `formatTimePeriod` function is used only within the context of a filter.

Example 1:

```
count(
                filter(
                                get("Account.dates"),
                                "maturity",
                               formatTimePeriod("day"),
                               equalTo(15)
                )
) > 3
```

This example returns true if there are more than three maturity dates that occur on the 15th day of the month.

Note:
See the time function for the supported time periods.

Example 2:

```
count(
                filter(
                                get("Account.dates"),
                                "maturity",
                               formatTimePeriod("month"),
                               equalTo(12)
                )
) > 3
```

This example returns true if there are more than three maturity dates in December (the 12th month).

get

Returns the value of one or more attribute values or lists. For a complex attribute, the `get` function returns a list of complex attribute records.

Specifying Attributes

To specify an attribute when you type a rule expression, use the syntax:

```
 owner.attribute_category.attribute
```

where the *owner* can be any of the following:

- Channel
- Corporate
- Customer
- Marketing
- Messaging

For example, an attribute might be identified Customer.demographics.gender where Customer is the owner, demographics is the attribute category, and gender is the attribute.

The following rule expression returns true when the value of the gender attribute is female.

```
get(
   "Customer.demographics.gender"
) == "Female"
```

Specifying Complex Attribute

A complex attribute category identifies groups of attributes. To specify a complex attribute category for use in rule expressions, use the syntax:

```
Account.complex_attribute_category_name
```

You specify the name of the complex attribute category, not the names of the attributes within the complex attribute category. For example, you might have configured a complex attribute category named cart that has two attributes: product and price. When the `get` function returns the contents of Account.cart, the cart complex attribute category can contain any number of groups of product attribute and price attribute.

The following rule expression returns true when first value for the product attribute in the cart complex attribute category is ToyTruck.

```
get(
   get("Account.cart", 0)
    "product"
) == "ToyTruck"
```

Session Modifier for Accessing Session Data

By default, the `get` function accesses the most current data. The `Session` modifier enables the `get` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session.

For example, suppose that you are tracking the name of each page visited by a customer on your website using a Customer.nav.page attribute. When each page is visited, you can update this attribute value with the page name. At any point in time, this value can be checked to get its current value by using a function like: `get("Customer.nav.page")`. Suppose that the customer first viewed the home page, which has an attribute value of `home`. Second, the customer viewed the login page, which has an attribute value of `login`. If you want to know which pages the customer visited within this session, or the specific sequence or pages visited, then you can use the `Session` keyword to see that information.

The following returns the current value of the page attribute. In our example, this returns `login`.

```
get(get("Session.Customer.nav.page"),0)
```

Changing the 0 to 1 returns the first value before the current value in the session date. In our example, this returns `home`.

```
get(get("Session.Customer.nav.page"),1)
```

The following rule expression returns true when the value for the Customer.nav.page attribute is `HOME`. The expression `get("Customer.nav.page")` with no parameters returns the system's current value for the attribute.

```
get(
   "Customer.nav.page"
) == "HOME"
```

The following rule expression returns true when the value for the Customer.nav.page attribute is HOME followed by the value CATALOG. The expression `get("Session.Customer.nav.page")` returns of all of the values for that attribute that have occurred in the entire session.

```
get(
   "Session.Customer.nav.page",
   0
 ) == "HOME"
 &&
 get(
   "Session.Customer.nav.page",
```

```
    1
) == "CATALOG"
```

In addition to the `Session` modifier, there are four parameters you can you use with the `get` function: Message, SessionExtensions, SessionResponses, and SessionServings.

Message Parameter

The function `get("Message")`returns the message that is currently being evaluated by the Teradata RTIM rules engine and includes the following for use in rules:

- `ID` (message ID)
- `name` (message name)

The following rule expression gets the current message and returns true when the name is special_offer.

```
get(
   get("Message"),
    "name"
) == "special_offer"
```

The following rule expression sets M to the message key name and returns true when that name is **message1**:

```
set("M", get("Message")) &&
get(M, "name") == "message1"
```

SessionExtensions Parameter

The records returned by `get("SessionExtensions")` represent an extension of a message by Teradata RTIM and include the following for use in rules:

- `ID` (message ID)
- `messageClass` (message group)
- `channel`
- `arbitration`
- `arbitrationValue`
- `strategy`
- `objective`

The following rule expression checks whether Teradata RTIM has extended the message to channel **c1** one or more times in the session.

```
count(filter(get("SessionExtensions"), "channel", "c1")) > 1
```

SessionResponses Parameter

The records returned by `get("SessionResponses")` represent responses to messages extended by Teradata RTIM and include the following for use in rules:

- `ID` (message ID)
- `channel`
- `interactionPoint`
- `responseLevel`

- `responseIsPositive` returns true or false (case sensitive)
- `convertedResponseTime`
- `implyExtension`
- `objective`
- `strategy`
- `messageClass` (message group)
- `strategySequence`
- `arbitration`
- `arbitrationValue`
- `name` (message name)

The following rule expression checks whether there have been two or more responses to the message named MSG1.

```
count(
    filter(
        filter(
            get("SessionResponses"),
             "name",
             "MSG1"
        ),
        "responseLevel",
    )
) > 2
```

In the following example, the rule expression is looking for two or more particular responses (accept) to the message named MSG1.

```
count(
   filter(
      filter(
         get("SessionResponses"),
          "name",
          "MSG1"
      ),
      "responseLevel",
      "accept"
   )
) > 2
```

SessionServings Parameter

The records returned by `get("SessionServings")` represent serving of a message by Teradata RTIM when Imply Extension is set to false and include the following for use in rules:

- `ID` (message ID)
- `messageClass` (message group)
- `channel`
- `arbitration`
- `arbitrationValue`
- `strategy`
- `objective`

The following rule expression checks whether Teradata RTIM has served the message to channel c1 one or more times in the session.

```
count(filter(get("SessionServings"), "channel", "c1")) > 1
```

greaterThan

Returns true for numerical attribute value greater than a given value. The `greaterThan` function is used only within the context of the `filter` or `is` functions.

In the following rule expression, the `greaterThan` function returns true when the cart complex attribute category has two items where the value for the price attribute is greater than `1.25`.

```
count(
    filter(
       get("Account.cart"),
       "price",
       greaterThan(1.25)
    )
) == 2
```

greaterThanOrEqual

Returns all the values greater than or equal to the given parameter values. The `greaterThanOrEqual` function is used only within the context of the `filter` or `is` functions.

```
count(
                filter(
                                get("Account.cart"),
                                "price",
                                greaterThanOrEqual(40)
                )
) == 3
```

The example returns true if there are three rows for **price** that have a value greater than or equal to 40.

index

Returns the index of the entry that matches internal filter function.

The following rule expression always returns true.

```
index(
   list("a", "b", "c"),
    "b"
) == 1
```

This is a zero-based index so a match on the first entry returns an index of `0`. The `index` function returns a `-1` when the specified value was not found.

is

Checks whether a given value matches another.

```
is(
     get("Customer.demographics.age"),
     between(25,50)
)
```

This expression evaluates to true when the customer's age is between 25 and 50.

lessThan

Returns true for numerical attribute value less than a given value. The `lessThan` function is used only within the context of the `filter` or `is` functions.

In the following rule expression, the `lessThan` function returns true when the cart complex attribute has two items where the value for the price attribute is less than 1.25.

```
count(
    filter(
       get("Account.cart"),
       "price",
       lessThan(1.25)
    )
) == 2
```

lessThanOrEqual

Returns all the values greater than or equal to the given parameter values. The `lessThanOrEqual` function is used only within the context of the `filter` or `is` functions.

```
count(
                filter(
                                get("Account.cart"),
                                "price",
                                lessThanOrEqual(40)
                )
) == 3
```

The example returns true if there are three rows for price that have a value less than or equal to 40.

list

Returns a temporary list of items.

The following rule expression creates the list a, b, c and always returns true, because the item positioned second in the list (represented by 1) is b.

```
get(
   list("a", "b", "c"),
```

```
    1
) == "b"
```

max

Returns the maximum value from a list of values.

In the following example, the rule expression returns true if the maximum value for any price attribute in the complex attribute category named cart is greater than 100.

```
max(
    get("Account.cart"),
     "price"
) > 100
```

If there are no entries for the price attribute, the `max` function returns a `0`.

By default, the `max` function accesses the most current data. The `Session` modifier enables the `max` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute with the `max` function.

Note:
You can use the `max` function with dates.

min

Returns the minimum value from a list of values.

In the following example, the rule expression returns true if the minimum price in the cart complex attribute category is greater than 12.

```
min(
    get("Account.cart"),
     "price"
) > 12
```

If there are no entries for the price attribute , the `min` function returns `0`.

By default, the `min` function accesses the most current data. The `Session` modifier enables the `min` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute with the `min` function.

Note:
You can use the `min` function with dates.

NaN

Returns true if the attribute value that should have been a mathematical number is not a number.

The following rule expression always returns true because an invalid average—the average of nothing—is not a number.

```
avg("") == NaN
```

Using the `value` function on a text string produces an error rather than `NaN` (not a number).

notEqualTo

Returns all the values not equal to the given parameter values. The `notEqualTo` function is used only within the context of the `filter` or `is` functions.

```
count(
                filter(
                                get("Account.cart"),
                                "price",
                                notEqualTo(40)
                )
) == 3
```

The example returns true if there are three rows for price that have a value not equal to 40.

now

Returns the current datetime (expressed as time in milliseconds since 1970/1/1 00:00:00 GMT), which can in turn be passed to the time function to be used for date math.

In the following rule expression, the `now` function is used with date formatting. The expression returns true when the current day is January 17th.

```
formatDate(now, "MM-dd") == "01-17"
```

parseDate

Takes a formatted date string and converts it to milliseconds since 1970/1/1 00:00:00 GMT.

```
parseDate(get("Customer.demographics.dob"))
```

The following example tests if a customer's date of birth is less than 30 days from the current time.

```
time(now) - time(parseDate(get("Customer.demographics.dob"))) < 86400000 * 30
```

The previous formula uses 86400000 because that is the number of milliseconds in one day. Alternatively, you can include the math that calculates the number of milliseconds in one day, as shown in the following example.

```
time(now) - time(parseDate(get("Customer.demographics.dob"))) < 1000 * 60 * 60
* 24 * 30
```

pattern

Creates a regular expression.

The pattern function uses the Java Pattern class to define a regular expression. The `pattern` function can include a test value for evaluation or can be used in conjunction with the `filter` and `index` functions. The optional second parameter specifies the flags defined by the Java Pattern class. The optional third parameter specifies a test value.

The following rule expression always returns true. (The second parameter in the following example—the number 2—is a flag for the Java Pattern class that indicates case-insensitivity. Do not confuse this flag with the use of the number 2 in most of the other rule expression functions, which would indicate a third item.)

```
pattern("Chris .*", 2, "chris Smith")
```

The following are examples of regular expressions you can use with the `pattern` function. The value for the second parameter has meaning but is not required.

```
pattern("Candy.*", 2, "CandyBar")                    // starts with
pattern("^(?!Bar).*", 2, " CandyBar ")            // not starts with
pattern(".*Bar",  2, " CandyBar ")                    // ends with
pattern("^(?!.*Candy$).*", 2, " CandyBar ")   // not ends with
pattern(".*Candy.*", 2, " CandyBar ")                // contains
pattern("^((?!cb).)*", 2, " CandyBar ")          // not contains
```

The following rule expression returns true if the cart complex attribute category includes a product attribute with a value that ends with Truck.

```
count(
  filter(
     get("Account.cart"),
     "product",
     pattern(".*Truck")
   )
 ) != 0
```

range

Returns true if the middle value is greater than or equal to the first parameter and less than or equal to the third parameter.

The following shows the syntax for `range`.

```
range (lower_value,  middle_value,  upper_value)
```

For example, the following rule expression returns true if a Customer.nav.page attribute with a value that has a pattern of .Credit.* is present within the last four instances of this attribute.

```
range(
    0,
    index(
        get("Session.Customer.nav.page"),
        pattern(".Credit.*")
    ),
    3
)
```

select

Gets value for a single attribute. If the attribute is a complex attribute category's attribute, then the `select` function can get all values for that attribute within the complex attribute's groups of attributes.

The following rule expression returns true if the cart complex attribute contains three product attributes.

```
count(select("product", get("Account.cart"))) == 3
```

set

Creates a temporary variable that can be used later within the current rule expression.

The following example sets the variable X to a value of B, so the rule expression always returns true.

```
set("X", "B") && X == "B"
```

sum

Returns the sum of a set of values.

The following example returns true if the sum of the price attributes in the complex attribute category named cart is greater than 1.

```
sum(
    get("Account.cart"),
     "price"
) > 1
```

By default, the `sum` function accesses the most current data. The `Session` modifier enables the `sum` function to access the session data, which is all of the data in the active session. For example, the `Session` modifier accesses not only the current value for an attribute but all the previous values for that attribute in the session. Use the `Session` modifier to use a simple attribute with the `sum` function.

time

Returns the time (as the number of milliseconds from the epoch date) from a datetime field. The `time` function is also used to convert a given datetime field into the specified time period. The function supports relative dates. The `time` function operates on the UTC time zone.

The `time` function supports the following time periods:

- hour: converts the given time to the hour of the day.

  For example, for `time(now,"hour")`, if `now` is 1:30 p.m., this resolves to today's date at 1 p.m.
- day: converts the given time to the day of the month.

  For example, for `time(now,"day")`, if `now` is the 5th day of the month, this resolves to today's date at zero hour (12:00 a.m.).
- week: converts the given time to the week of the year.

  For example, `time(now,"week")` resolves to a datetime that represents the week of the year at zero hour (12:00 a.m.).
- month: converts the given time to the month of the year.

For example, `time(now,"month")` resolves to a datetime that represents the month of the year at zero hour (12:00 a.m.).

- quarter: converts the given time to the month of the year representing the start of the quarter.

  For example, for `time(now,"quarter")`, if `now` is 12/31/2012, this resolves to 10/01/2012 at zero hour (12:00 a.m.).
- year: converts the given time to calendar year.

  For example, for `time(now,"year")`, if `now` is 12/31/2012, this resolves to 01/01/2012 at zero hour (12:00 a.m.).

Example 1:

As shown in the following example, you can pass the result of parseDate to the `time` function.

```
now - time(
    parseDate(
        get("Customer.demographics.dob")
    )
) < 1000 * 60 * 60 * 24 * 30
```

Example 2:

The following expression evaluates to true if today is May 2015 regardless of the day and time.

```
time(now,"month") ==  time("2015-05-11T08:20:00.000Z","month")
```

Example 3:

The `time` function also supports relative dates and time periods. For example, `time(now,5,"day","past")` calculates the time 5 days in the past from the given time. If `now` is 11/24/2014, this returns 11/19/2014 at zero hour (12:00 a.m.). For another example, `time(now,12,"hour","future")` calculates the time 12 hours in the future. If `now` is 9 a.m., this returns 9 p.m.

The following expression evaluates to true if the expiration date occurs before 3 months from now.

```
time(get("Account.dates.expiration ")) < time(now,3,"month","future")
```

value

Converts a string value into a number.

When a number is passed in as a string, the `value` function causes the value to be treated as a number.

The following rule expression returns true if the dog is 10 years old (that is, the Customer.dog.age attribute has a value of 10), because 7 multiplied by 10 equals 70.

```
7 * value(get("Customer.dog.age")) == 70
```

Combining Rule Expression Functions

Functions can be nested inside each other. You can also chain sets of functions together to create more complex rule expressions. All standard rules for Java functions are supported such as the && (AND) operator and the || (OR) operator.

The following rule expression sets a temporary variable called filteredCart with the list of cart complex attribute category's item attributes that have a value that matches ItemX. The rule expression then creates a temporary variable called filteredWallet with a list of wallet discounted items that match ItemX.

Finally, the rule expression returns true if the count of items in the filteredCart is greater than the number of items in the filteredWallet.

```
set("filteredCart",
  filter(
      get("Account.cart"),"item","ItemX"
  )
)&&
set("filteredWallet",
  filter(
      get("Account.wallet"),"discountedItem","ItemX"
  )
)&&
count(filteredCart) > count(filteredWallet)
```

The following rule expression uses the `pattern` function extensively as well as the && operator. The rule expression returns true if the customer has never seen a Savings page and has seen Product and Cards pages and Products within the last two pages and is currently on the Credit page.

```
index(
     get("Session.Customer.nav.page"),
     pattern(".*Savings")
) == -1 &&
index(
     get("Session.Customer.nav.page"),
     list(
        pattern("Products.*"),
        pattern(".*Cards.*"),
        pattern("Products.*", 2)
    )
) > -1 &&
get("Customer.nav.page") == "Credit"
```