# Hortonworks DataFlow

## Expression Language

(June 6, 2018)

docs.cloudera.com

# Hortonworks DataFlow: Expression Language

# Table of Contents

# List of Tables

# 1. Apache NiFi Expression Language Guide

## 1.1. Apache NiFi Expression Language Guide

Table of Contents

# Overview

All data in Apache NiFi is represented by an abstraction called a FlowFile. A FlowFile is comprised of two major pieces: content and attributes. The content portion of the FlowFile represents the data on which to operate. For instance, if a file is picked up from a local file system using the GetFile Processor, the contents of the file will become the contents of the FlowFile.

The attributes portion of the FlowFile represents information about the data itself, or metadata. Attributes are key-value pairs that represent what is known about the data as well as information that is useful for routing and processing the data appropriately. Keeping with the example of a file that is picked up from a local file system, the FlowFile would have an attribute called `filename` that reflected the name of the file on the file system. Additionally, the FlowFile will have a `path` attribute that reflects the directory on the file system that this file lived in. The FlowFile will also have an attribute named `uuid`, which is a unique identifier for this FlowFile. For complete listing of the core attributes check out the FlowFile section of the Developer's Guide.

However, placing these attributes on a FlowFile do not provide much benefit if the user is unable to make use of them. The NiFi Expression Language provides the ability to reference these attributes, compare them to other values, and manipulate their values.

## 1.1.1. Structure of a NiFi Expression

The NiFi Expression Language always begins with the start delimiter `${` and ends with the end delimiter `}`. Between the start and end delimiters is the text of the Expression itself. In its most basic form, the Expression can consist of just an attribute name. For example, `${filename}` will return the value of the ``filename" attribute.

In a slightly more complex example, we can instead return a manipulation of this value. We can, for example, return an all upper-case version of the filename by calling the `toUpper` function: `${filename:toUpper()}`. In this case, we reference the `filename'' attribute and then manipulate this value by using the toUpper function. A function call consists of 5 elements. First, there is a function call delimiter :. Second is the name of the function - in this case, toUpper"`. Next is an open parenthesis (`(`), followed by the function arguments. The arguments necessary are dependent upon which function is being called. In this example, we are using the `toUpper` function, which does not have any arguments, so this element is omitted. Finally, the closing parenthesis (`)`) indicates the end of the function call. There are many different functions that are supported by the Expression Language to achieve many different goals. Some functions provide String (text) manipulation, such as the `toUpper` function. Others, such as the `equals` and `matches` functions, provide comparison functionality. Functions also exist for manipulating dates and times and for performing mathematical operations. Each of these functions is described below, in the Functions section, with an explanation of what the function does, the arguments that it requires, and the type of information that it returns.

When we perform a function call on an attribute, as above, we refer to the attribute as the *subject* of the function, as the attribute is the entity on which the function is operating. We can then chain together multiple function calls, where the return value of the first function becomes the subject of the second function and its return value becomes the subject of the third function and so on. Continuing with our example, we can chain together multiple functions by using the expression `${filename:toUpper():equals('HELLO.TXT')}`. There is no limit to the number of functions that can be chained together.

Any FlowFile attribute can be referenced using the Expression Language. However, if the attribute name contains a `special character,'' the attribute name must be escaped by quoting it. The following characters are each considered` special characters":

- $ (dollar sign)
- | (pipe)
- { (open brace)
- } (close brace)
- ( (open parenthesis)

- ) (close parenthesis)
- [ (open bracket)
- ] (close bracket)
- , (comma)
- : (colon)
- ; (semicolon)
- / (forward slash)
- * (asterisk)
- ' (single quote)
- (space)
- \t (tab)
- \r (carriage return)
- \n (new-line)

Additionally, a number is considered a `special character'' if it is the first character of the attribute name. If any of these special characters is present in an attribute is quoted by using either single or double quotes. The Expression Language allows single quotes and double quotes to be used interchangeably. For example, the following can be used to escape an attribute named` my attribute": `${"my attribute"}` or `${'my attribute'}`.

In this example, the value to be returned is the value of the "my attribute" value, if it exists. If that attribute does not exist, the Expression Language will then look for a System Environment Variable named "my attribute." If unable to find this, it will look for a JVM System Property named "my attribute." Finally, if none of these exists, the Expression Language will return a `null` value.

There also exist some functions that expect to have no subject. These functions are invoked simply by calling the function at the beginning of the Expression, such as `${hostname()}`. These functions can then be changed together, as well. For example, `${hostname():toUpper()}`. Attempting to evaluate the function with subject will result in an error. In the Functions section below, these functions will clearly indicate in their descriptions that they do not require a subject.

Often times, we will need to compare the values of two different attributes to each other. We are able to accomplish this by using embedded Expressions. We can, for example, check if the `filename'' attribute is the same as the` uuid" attribute: `${filename:equals( ${uuid} )}`. Notice here, also, that we have a space between the opening parenthesis for the `equals` method and the embedded Expression. This is not necessary and does not affect how the Expression is evaluated in any way. Rather, it is

intended to make the Expression easier to read. White space is ignored by the Expression Language between delimiters. Therefore, we can use the Expression `${ filename : equals(${ uuid}) }` or `${filename:equals(${uuid})}` and both Expressions mean the same thing. We cannot, however, use `${file name:equals(${uuid})}`, because this results in `file` and `name` being interpreted as different tokens, rather than a single token, `filename`.

## 1.1.2. Expression Language in the Application

The Expression Language is used heavily throughout the NiFi application for configuring Processor properties. Not all Processor properties support the Expression Language, however. Whether or not a Property supports the Expression Language is determined by the developer of the Processor when the Processor is written. However, the application strives to clearly illustrate for each Property whether or not the Expression Language is supported.

In the application, when configuring a Processor property,
the User Interface provides an Information icon (

) next to the name of the Property. Hovering over this icon with the mouse will provide a tooltip that provides helpful information about the Property. This information includes a description of the Property, the default value (if any), historically configured values (if any), and whether or not this Property supports the expression language.

### 1.1.2.1. Expression Language Editor

When configuring the value of a Processor property, the NiFi User Interface provides help with the Expression Language using the Expression Language editor. Once an Expression is begin by typing `${`, the editor begins to highlight parentheses and braces so that the user is easily able to tell which opening parenthesis or brace matches which closing parenthesis or brace.

The editor also supplies context-sensitive help by providing a list of all functions that can be used at the current cursor position. To activate this feature, press Ctrl+Space on the keyboard. The user is also able to type part of a function name and then press Ctrl+Space to see all functions that can be used that start with the same prefix. For example, if we type into the editor `${filename:to` and then press Ctrl+Space, we are provided a pop-up that lists six different functions: `toDate`, `toLower`, `toNumber`, `toRadix`, `toString`, and `toUpper`. We can then continue typing to narrow which functions are shown, or we can select one of the functions from the list by double-clicking it with the mouse or using the arrow keys to highlight the desired function and pressing Enter.

## 1.1.3. Functions

Functions provide a convenient way to manipulate and compare values of attributes. The Expression Language provides many different functions to meet the needs of a automated dataflow. Each function takes zero or more arguments and returns a single value. These functions can then be chained together to create powerful Expressions to evaluate conditions and manipulate values. See Structure of a NiFi Expression for more information on how to call and chain functions together.

#### 1.1.3.1. Data Types

Each argument to a function and each value returned from a function has a specific data type. The Expression Language supports four different data types:

- **String**: A String is a sequence of characters that can consist of numbers, letters, white space, and special characters.
- **Number**: A Number is an whole number comprised of one or more digits (`0` through `9`). When converting to numbers from Date data types, they are represented as the number of milliseconds since midnight GMT on January 1, 1970.
- **Decimal**: A Decimal is a numeric value that can support decimals and larger values with minimal loss of precision. More precisely it is a double-precision 64-bit IEEE 754 floating point. Due to this minimal loss of precision this data type should not be used for very precise values, such as currency. For more documentation on the range of values stored in this data type refer to this link. The following are some examples of the forms of literal decimals that are supported in expression language (the "E" can also be lower-case):
  - 1.1
  - .1E1
  - 1.11E-12
- **Date**: A Date is an object that holds a Date and Time. Utilizing the Date Manipulation and Type Coercion functions this data type can be converted to/from Strings and numbers. If the whole Expression Language expression is evaluated to be a date then it will be converted to a String with the format: "<Day of Week> <Month> <Day of Month> <Hour>:<Minute>:<Second> <Time Zone> <Year>". Also expressed as "E MMM dd HH:mm:ss z yyyy" in Java SimpleDateFormat format. For example: "Wed Dec 31 12:00:04 UTC 2016".
- **Boolean**: A Boolean is one of either `true` or `false`.

After evaluating expression language functions, all attributes are stored as type String.

The Expression Language is generally able to automatically coerce a value of one data type to the appropriate data type for a function. However, functions do exist to manually coerce a value into a specific data type. See the Type Coercion section for more information.

Hex values are supported for Number and Decimal types but they must be quoted and prepended with "0x" when being interpreted as literals. For example these two expressions are valid (without the quotes or "0x" the expression would fail to run properly):

- ${literal("0xF"):toNumber()}
- ${literal("0xF.Fp10"):toDecimal()}

## 1.1.4. Boolean Logic

One of the most powerful features of the Expression Language is the ability to compare an attribute value against some other value. This is used often, for example, to configure how

a Processor should route data. The following functions are used for performing boolean logic, such as comparing two values. Each of these functions are designed to work on values of type Boolean.

#### 1.1.4.1. isNull

**Description**: The `isNull` function returns `true` if the subject is null, `false` otherwise. This is typically used to determine if an attribute exists.

**Subject Type**: Any

**Arguments**: No arguments

**Return Type**: Boolean

**Examples**: `${filename:isNull()}` returns `true` if the "filename" attribute does not exist. It returns `false` if the attribute exists.

#### 1.1.4.2. notNull

**Description**: The `notNull` function returns the opposite value of the `isNull` function. That is, it will return `true` if the subject exists and `false` otherwise.

**Subject Type**: Any

**Arguments**: No arguments

**Return Type**: Boolean

**Examples**: `${filename:notNull()}` returns `true` if the "filename" attribute exists. It returns `false` if the attribute does not exist.

#### 1.1.4.3. isEmpty

**Description**: The `isEmpty` function returns `true` if the Subject is null, does not contain any characters or contains only white-space (new line, carriage return, space, tab), `false` otherwise.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: Boolean

**Examples**: `${filename:isEmpty()}` returns `true` if the "filename" attribute does not exist or contains only white space. `${literal(""):isEmpty()}` returns true as well as `${literal(""):isEmpty()}`.

#### 1.1.4.4. equals

**Description**: The `equals` function is very widely used and determines if its subject is equal to another String value. Note that the `equals` function performs a direct comparison of

two String values. Take care not to confuse this function with the matches function, which evaluates its subject against a Regular Expression.

**Subject Type**: Any

**Arguments**:

- *value* : The value to compare the Subject to. Must be same type as the Subject.

**Return Type**: Boolean

**Examples**: We can check if the filename of a FlowFile is "hello.txt" by using the expression `${filename:equals('hello.txt')}`, or we could check if the value of the attribute `hello` is equal to the value of the `filename` attribute: `${hello:equals( ${filename} )}`.

### 1.1.4.5. equalsIgnoreCase

**Description**: Similar to the `equals` function, the `equalsIgnoreCase` function compares its subject against a String value but returns `true` if the two values differ only by case (upper case vs. lower case).

**Subject Type**: String

**Arguments**:

- *value* : The value to compare the Subject to.

**Return Type**: Boolean

**Examples**: `${filename:equalsIgnoreCase('hello.txt')}` will evaluate to `true` if filename is equal to "hello.txt" or "HELLO.TXT" or "HeLLo.TxT".

### 1.1.4.6. gt

**Description**: The `gt` function is used for numeric comparison and returns `true` if the subject is Greater Than its argument. If either the subject or the argument cannot be coerced into a Number, this function returns `false`.

**Subject Type**: Number

**Arguments**:

- *value* : The number to compare the Subject to.

**Return Type**: Boolean

**Examples**: `${fileSize:gt( 1024 )}` will return `true` if the size of the FlowFile's content is more than 1 kilobyte (1024 bytes). Otherwise, it will return `false`.

### 1.1.4.7. ge

**Description**: The `ge` function is used for numeric comparison and returns `true` if the subject is Greater Than Or Equal To its argument. If either the subject or the argument cannot be coerced into a Number, this function returns `false`.

**Subject Type**: Number

**Arguments**:

- *value* : The number to compare the Subject to.

**Return Type**: Boolean

**Examples**: `${fileSize:ge( 1024 )}` will return `true` if the size of the FlowFile's content is at least ( is greater than or equal to) 1 kilobyte (1024 bytes). Otherwise, it will return `false`.

### 1.1.4.8. lt

**Description**: The `lt` function is used for numeric comparison and returns `true` if the subject is Less Than its argument. If either the subject or the argument cannot be coerced into a Number, this function returns `false`.

**Subject Type**: Number

**Arguments**:

- *value* : The number to compare the Subject to.

**Return Type**: Boolean

**Examples**: `${fileSize:lt( 1048576 )}` will return `true` if the size of the FlowFile's content is less than 1 megabyte (1048576 bytes). Otherwise, it will return `false`.

### 1.1.4.9. le

**Description**: The `le` function is used for numeric comparison and returns `true` if the subject is Less Than Or Equal To its argument. If either the subject or the argument cannot be coerced into a Number, this function returns `false`.

**Subject Type**: Number

**Arguments**:

- *value* : The number to compare the Subject to.

**Return Type**: Boolean

**Examples**: `${fileSize:le( 1048576 )}` will return `true` if the size of the FlowFile's content is at most (less than or equal to) 1 megabyte (1048576 bytes). Otherwise, it will return `false`.

### 1.1.4.10. and

**Description**: The `and` function takes as a single argument a Boolean value and returns `true` if both the Subject and the argument are `true`. If either the subject or the argument is `false` or cannot be coerced into a Boolean, the function returns `false`. Typically, this is used with an embedded Expression as the argument.

**Subject Type**: Boolean

**Arguments**:

- *condition* : The right-hand-side of the 'and' Expression

**Return Type**: Boolean

**Examples**: We can check if the filename is both all lower-case and has at least 5 characters by using the Expression

```
${filename:toLower():equals( ${filename} ):and(
        ${filename:length():ge(5)}
)}
```

### 1.1.4.11. or

**Description**: The `or` function takes as a single argument a Boolean value and returns `true` if either the Subject or the argument is `true`. If both the subject and the argument are `false`, the function returns `false`. If either the Subject or the argument cannot be coerced into a Boolean value, this function will return `false`.

**Subject Type**: Boolean

**Arguments**:

- *condition* : The right-hand-side of the 'and' Expression

**Return Type**: Boolean

**Examples**: The following example will return `true` if either the filename has exactly 5 characters or if the filename is all lower-case.

```
${filename:toLower():equals( ${filename} ):or(
        ${filename:length():equals(5)}
)}
```

### 1.1.4.12. not

**Description**: The `not` function returns the negation of the Boolean value of the subject.

**Subject Type**: Boolean

**Arguments**: No arguments

**Return Type**: Boolean

**Examples**: We can invert the value of another function by using the `not` function, as `${filename:equals('hello.txt'):not()}`. This will return `true` if the filename is NOT equal to "hello.txt" and will return `false` if the filename is "hello.txt."

### 1.1.4.13. ifElse

**Description**: Evaluates the first argument if the Subject evaluates to true, or the second argument if the Subject evaluates to false.

**Subject Type**: Boolean

**Arguments**:

- *EvaluateIfTrue* : The value to return if the Subject is true
- *EvaluateIfFalse* : The value to return if the Subject is false

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", the "nullFilename" attribute has the value null, and the "bool" attribute has the value "true", then the following expressions will provide the following results:

**Table 1.1. Table 1. ifElse Examples**

| Expression | Value |
|---|---|
| `${bool:ifElse('a','b')}` | a |
| `${literal(true):ifElse('a','b')}` | a |
| `${nullFilename:isNull():ifElse('file does not exist', 'located file')}` | file does not exist |
| `${nullFilename:ifElse('found', 'not_found')}` | not_found |
| `${filename:ifElse('found', 'not_found')}` | not_found |
| `${filename:isNull():not():ifElse('found', 'not_found')}` | found |

## 1.1.5. String Manipulation

Each of the following functions manipulates a String in some way.

### 1.1.5.1. toUpper

**Description**: This function converts the Subject into an all upper-case String. Said another way, it replaces any lowercase letter with the uppercase equivalent.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "filename" attribute is "abc123.txt", then the Expression `${filename:toUpper()}` will return "ABC123.TXT"

### 1.1.5.2. toLower

**Description**: This function converts the Subject into an all lower-case String. Said another way, it replaces any uppercase letter with the lowercase equivalent.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "filename" attribute is "ABC123.TXT", then the Expression `${filename:toLower()}` will return "abc123.txt"

### 1.1.5.3. trim

**Description**: The `trim` function will remove any leading or trailing white space from its subject.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the attribute `attr` has the value " 1 2 3 ", then the Expression `${attr:trim()}` will return the value "1 2 3".

### 1.1.5.4. substring

**Description**: Returns a portion of the Subject, given a *starting index* and an optional *ending index*. If the *ending index* is not supplied, it will return the portion of the Subject starting at the given 'start index' and ending at the end of the Subject value.

The *starting index* and *ending index* are zero-based. That is, the first character is referenced by using the value `0`, not `1`.

If either the *starting index* is or the *ending index* is not a number, this function call will result in an error.

If the *starting index* is larger than the *ending index*, this function call will result in an error.

If the *starting index* or the *ending index* is greater than the length of the Subject or has a value less than 0, this function call will result in an error.

**Subject Type**: String

**Arguments**:

- *starting index* : The 0-based index of the first character to capture (inclusive)
- *ending index* : The 0-based index of the last character to capture (exclusive)

**Return Type**: String

**Examples**:

If we have an attribute named "filename" with the value "a brand new filename.txt", then the following Expressions will result in the following values:

#### Table 1.2. Table 2. Substring Examples

| Expression | Value |
|---|---|

| | |
|---|---|
| `${filename:substring(0,1)}` | `a` |
| `${filename:substring(2)}` | `brand new filename.txt` |
| `${filename:substring(12)}` | `filename.txt` |
| `${filename:substring( ${filename:length():minus(2)} )}` | `xt` |

### 1.1.5.5. substringBefore

**Description**: Returns a portion of the Subject, starting with the first character of the Subject and ending with the character immediately before the first occurrence of the argument. If the argument is not present in the Subject, the entire Subject will be returned.

**Subject Type**: String

**Arguments**:

- *value* : The String to search for in the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will result in the following values:

#### Table 1.3. Table 3. SubstringBefore Examples

| Expression | Value |
|---|---|
| `${filename:substringBefore('.')}` | `a brand new filename` |
| `${filename:substringBefore('')}` | `a` |
| `${filename:substringBefore(' n')}` | `a brand` |
| `${filename:substringBefore('missing')}` | `a brand new filename.txt` |

### 1.1.5.6. substringBeforeLast

**Description**: Returns a portion of the Subject, starting with the first character of the Subject and ending with the character immediately before the last occurrence of the argument. If the argument is not present in the Subject, the entire Subject will be returned.

**Subject Type**: String

**Arguments**:

- *value* : The String to search for in the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will result in the following values:

#### Table 1.4. Table 4. SubstringBeforeLast Examples

| Expression | Value |
|---|---|
| `${filename:substringBeforeLast('.')}` | `a brand new filename` |

| | |
|---|---|
| `${filename:substringBeforeLast('')}` | `a brand new` |
| `${filename:substringBeforeLast(' n')}` | `a brand` |
| `${filename:substringBeforeLast('missing')}` | `a brand new filename.txt` |

### 1.1.5.7. substringAfter

**Description**: Returns a portion of the Subject, starting with the character immediately after the first occurrence of the argument and extending to the end of the Subject. If the argument is not present in the Subject, the entire Subject will be returned.

**Subject Type**: String

**Arguments**:

- *value* : The String to search for in the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will result in the following values:

#### Table 1.5. Table 5. SubstringAfter Examples

| Expression | Value |
|---|---|
| `${filename:substringAfter('.')}` | `txt` |
| `${filename:substringAfter('')}` | `brand new filename.txt` |
| `${filename:substringAfter(' n')}` | `ew filename.txt` |
| `${filename:substringAfter('missing')}` | `a brand new filename.txt` |

### 1.1.5.8. substringAfterLast

**Description**: Returns a portion of the Subject, starting with the character immediately after the last occurrence of the argument and extending to the end of the Subject. If the argument is not present in the Subject, the entire Subject will be returned.

**Subject Type**: String

**Arguments**:

- *value* : The String to search for in the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will result in the following values:

#### Table 1.6. Table 6. SubstringAfterLast Examples

| Expression | Value |
|---|---|
| `${filename:substringAfterLast('.')}` | `txt` |
| `${filename:substringAfterLast('')}` | `filename.txt` |

| | |
|---|---|
| `${filename:substringAfterLast(' n')}` | `ew filename.txt` |
| `${filename:substringAfterLast('missing')}` | `a brand new filename.txt` |

### 1.1.5.9. getDelimitedField

**Description**: Parses the Subject as a delimited line of text and returns just a single field from that delimited text.

**Subject Type**: String

**Arguments**:

- *index* : The index of the field to return. A value of 1 will return the first field, a value of 2 will return the second field, and so on.
- *delimiter* : Optional argument that provides the character to use as a field separator. If not specified, a comma will be used. This value must be exactly 1 character.
- *quoteChar* : Optional argument that provides the character that can be used to quote values so that the delimiter can be used within a single field. If not specified, a double-quote (") will be used. This value must be exactly 1 character.
- *escapeChar* : Optional argument that provides the character that can be used to escape the Quote Character or the Delimiter within a field. If not specified, a backslash (\) is used. This value must be exactly 1 character.
- *stripChars* : Optional argument that specifies whether or not quote characters and escape characters should be stripped. For example, if we have a field value "1, 2, 3" and this value is true, we will get the value `1, 2, 3`, but if this value is false, we will get the value `"1, 2, 3"` with the quotes. The default value is false. This value must be either `true` or `false`.

**Return Type**: String

**Examples**: If the "line" attribute contains the value *"Jacobson, John", 32, Mr.* and the "altLine" attribute contains the value *Jacobson, John|32|Mr.* then the following Expressions will result in the following values:

#### Table 1.7. Table 7. GetDelimitedField Examples

| Expression | Value |
|---|---|
| `${line:getDelimitedField(2)}` | _(space)_32 |
| `${line:getDelimitedField(2):trim()}` | 32 |
| `${line:getDelimitedField(1)}` | "Jacobson, John" |
| `${line:getDelimitedField(1, ',', '"', '\ \', true)}` | Jacobson, John |
| `${altLine:getDelimitedField(1, '|')}` | Jacobson, John |

### 1.1.5.10. append

**Description**: The `append` function returns the result of appending the argument to the value of the Subject. If the Subject is null, returns the argument itself.

**Subject Type**: String

**Arguments**:

- *value* : The String to append to the end of the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the Expression `${filename:append('.gz')}` will return "a brand new filename.txt.gz".

### 1.1.5.11. prepend

**Description**: The `prepend` function returns the result of prepending the argument to the value of the Subject. If the subject is null, returns the argument itself.

**Subject Type**: String

**Arguments**:

- *value* : The String to prepend to the beginning of the Subject

**Return Type**: String

**Examples**: If the "filename" attribute has the value "filename.txt", then the Expression `${filename:prepend('a brand new ')}` will return "a brand new filename.txt".

### 1.1.5.12. replace

**Description**: Replaces **all** occurrences of one literal String within the Subject with another String.

**Subject Type**: String

**Arguments**:

- *Search String* : The String to find within the Subject
- *Replacement* : The value to replace *Search String* with

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

#### Table 1.8. Table 8. Replace Examples

| Expression | Value |
|---|---|
| `${filename:replace('.', '_')}` | `a brand new filename_txt` |
| `${filename:replace('', '.')}` | `a.brand.new.filename.txt` |
| `${filename:replace('XYZ', 'ZZZ')}` | `a brand new filename.txt` |
| `${filename:replace('filename', 'book')}` | `a brand new book.txt` |

### 1.1.5.13. replaceFirst

**Description**: Replaces **the first** occurrence of one literal String or regular expression within the Subject with another String.

**Subject Type**: String

**Arguments**:

- *Search String* : The String (literal or regular expression pattern) to find within the Subject
- *Replacement* : The value to replace *Search String* with

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

#### Table 1.9. Table 9. ReplaceFirst Examples

| Expression | Value |
|---|---|
| `${filename:replaceFirst('a', 'the')}` | `the brand new filename.txt` |
| `${filename:replaceFirst('[br]', 'g')}` | `a grand new filename.txt` |
| `${filename:replaceFirst('XYZ', 'ZZZ')}` | `a brand new filename.txt` |
| `${filename:replaceFirst('\w{8}', 'book')}` | `a brand new book.txt` |

### 1.1.5.14. replaceAll

**Description**: The `replaceAll` function takes two String arguments: a literal String or Regular Expression (NiFi uses the Java Pattern syntax), and a replacement string. The return value is the result of substituting the replacement string for all patterns within the Subject that match the Regular Expression.

**Subject Type**: String

**Arguments**:

**Arguments**:

- *Regex* : he Regular Expression (in Java syntax) to match in the Subject
- *Replacement* : The value to use for replacing matches in the Subject. If the *regular expression* argument uses Capturing Groups, back references are allowed in the *replacement*.

**Return Type**: String

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

#### Table 1.10. Table 10. ReplaceAll Examples

| Expression | Value |
|---|---|

| | |
|---|---|
| `${filename:replaceAll('\..*', '')}` | `a brand new filename` |
| `${filename:replaceAll('a brand (new)', '$1')}` | `new filename.txt` |
| `${filename:replaceAll('XYZ', 'ZZZ')}` | `a brand new filename.txt` |
| `${filename:replaceAll('brand (new)', 'somewhat $1')}` | `a somewhat new filename.txt` |

### 1.1.5.15. replaceNull

**Description**: The `replaceNull` function returns the argument if the Subject is null. Otherwise, returns the Subject.

**Subject Type**: Any

**Arguments**:

- *Replacement* : The value to return if the Subject is null.

**Return Type**: Type of Subject if Subject is not null; else, type of Argument

**Examples**: If the attribute "filename" has the value "a brand new filename.txt" and the attribute "hello" does not exist, then the Expression `${filename:replaceNull('abc')}` will return "a brand new filename.txt", while `${hello:replaceNull('abc')}` will return "abc".

### 1.1.5.16. replaceEmpty

**Description**: The `replaceEmpty` function returns the argument if the Subject is null or if the Subject consists only of white space (new line, carriage return, tab, space). Otherwise, returns the Subject.

**Subject Type**: String

**Arguments**:

- *Replacement* : The value to return if the Subject is null or empty.

**Return Type**: String

**Examples**: If the attribute "filename" has the value "a brand new filename.txt" and the attribute "hello" has the value "", then the Expression `${filename:replaceEmpty('abc')}` will return "a brand new filename.txt", while `${hello:replaceEmpty('abc')}` will return "abc".

### 1.1.5.17. length

**Description**: Returns the length of the Subject

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: Number

**Examples**: If the attribute "filename" has a value of "a brand new filename.txt" and the attribute "hello" does not exist, then the Expression `${filename:length()}` will return 24. `${hello:length()}` will return 0.

## 1.1.6. Encode/Decode Functions

Each of the following functions will encode a string according the rules of the given data format.

### 1.1.6.1. escapeJson

**Description**: This function prepares the Subject to be inserted into JSON document by escaping the characters in the String using Json String rules. The function correctly escapes quotes and control-chars (tab, backslash, cr, ff, etc.)

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is 'He didn't say, "Stop!"', then the Expression `${message:escapeJson()}` will return 'He didn't say, \"Stop!\"'

### 1.1.6.2. escapeXml

**Description**: This function prepares the Subject to be inserted into XML document by escaping the characters in a String using XML entities. The function correctly escapes quotes, apostrophe, ampersand, <, > and control-chars.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:escapeXml()}` will return '"bread" & "butter"'

### 1.1.6.3. escapeCsv

**Description**: This function prepares the Subject to be inserted into CSV document by escaping the characters in a String using the rules in RFC 4180. The function correctly escapes quotes and surround the string in quotes if needed.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is 'But finally, she left', then the Expression `${message:escapeCsv()}` will return '"But finally, she left"'

#### 1.1.6.4. escapeHtml3

**Description**: This function prepares the Subject to be inserted into HTML document by escaping the characters in a String using the HTML entities. Supports only the HTML 3.0 entities.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:escapeHtml3()}` will return '"bread" & "butter"'

#### 1.1.6.5. escapeHtml4

**Description**: This function prepares the Subject to be inserted into HTML document by escaping the characters in a String using the HTML entities. Supports all known HTML 4.0 entities.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:escapeHtml4()}` will return '"bread" & "butter"'

#### 1.1.6.6. unescapeJson

**Description**: This function unescapes any Json literals found in the String.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is 'He didn't say, \"Stop!\"', then the Expression `${message:unescapeJson()}` will return 'He didn't say, "Stop!"'

#### 1.1.6.7. unescapeXml

**Description**: This function unescapes a string containing XML entity escapes to a string containing the actual Unicode characters corresponding to the escapes. Supports only the five basic XML entities (gt, lt, quot, amp, apos).

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:unescapeXml()}` will return '"bread" & "butter"'

### 1.1.6.8. unescapeCsv

**Description**: This function unescapes a String from a CSV document according to the rules of RFC 4180.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"But finally, she left"', then the Expression `${message:unescapeCsv()}` will return 'But finally, she left'

### 1.1.6.9. unescapeHtml3

**Description**: This function unescapes a string containing HTML 3 entity to a string containing the actual Unicode characters corresponding to the escapes. Supports only HTML 3.0 entities.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:unescapeHtml3()}` will return '"bread" & "butter"'

### 1.1.6.10. unescapeHtml4

**Description**: This function unescapes a string containing HTML 4 entity to a string containing the actual Unicode characters corresponding to the escapes. Supports all known HTML 4.0 entities.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If the "message" attribute is '"bread" & "butter"', then the Expression `${message:unescapeHtml4()}` will return '"bread" & "butter"'

### 1.1.6.11. urlEncode

**Description**: Returns a URL-friendly version of the Subject. This is useful, for instance, when using an attribute value to indicate the URL of a website.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: We can URL-Encode an attribute named "url" by using the Expression `${url:urlEncode()}`. If the value of the "url" attribute is "https://nifi.apache.org/some value with spaces", this Expression will then return "https://nifi.apache.org/some%20value %20with%20spaces".

#### 1.1.6.12. urlDecode

**Description**: Converts a URL-friendly version of the Subject into a human-readable form.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If we have a URL-Encoded attribute named "url" with the value "https://nifi.apache.org/some%20value%20with%20spaces", then the Expression `${url:urlDecode()}` will return "https://nifi.apache.org/some value with spaces".

#### 1.1.6.13. base64Encode

**Description**: Returns a Base64 encoded string. This is useful for being able to transfer binary data as ascii.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: We can Base64-Encoded an attribute named "payload" by using the Expression `${payload:base64Encode()}` If the attribute payload had a value of "admin:admin" then the Expression `${payload:base64Encode()}` will return "YWRtaW46YWRtaW4=".

#### 1.1.6.14. base64Decode

**Description**: Reverses the Base64 encoding on given string.

**Subject Type**: String

**Arguments**: No arguments

**Return Type**: String

**Examples**: If we have a Base64-Encoded attribute named "payload" with the value "YWRtaW46YWRtaW4=", then the Expression `${payload:base64Decode()}` will return "admin:admin".

### 1.1.7. Searching

Each of the following functions is used to search its subject for some value.

#### 1.1.7.1. startsWith

**Description**: Returns `true` if the Subject starts with the String provided as the argument, `false` otherwise.

**Subject Type**: String

**Arguments**:

- *value* : The value to search for

**Return Type**: Boolean

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the Expression `${filename:startsWith('a brand')}` will return `true`. `${filename:startsWith('A BRAND')}` will return `false`. `${filename:toUpper():startsWith('A BRAND')}` returns `true`.

#### 1.1.7.2. endsWith

**Description**: Returns `true` if the Subject ends with the String provided as the argument, `false` otherwise.

**Subject Type**: String

**Arguments**:

- *value* : The value to search for

**Return Type**: Boolean

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the Expression `${filename:endsWith('txt')}` will return `true`. `${filename:endsWith('TXT')}` will return `false`. `${filename:toUpper():endsWith('TXT')}` returns `true`.

#### 1.1.7.3. contains

**Description**: Returns `true` if the Subject contains the value of the argument anywhere in the value.

**Subject Type**: String

**Arguments**:

- *value* : The value to search for

**Return Type**: Boolean

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the Expression `${filename:contains('new')}` will return `true`. `${filename:contains('NEW')}` will return `false`. `${filename:toUpper():contains('NEW')}` returns `true`.

### 1.1.7.4. in

**Description**: Returns `true` if the Subject is matching one of the provided arguments.

**Subject Type**: String

**Arguments**:

- *value1* : First possible matching value
- *valueN* : Nth possible matching value

**Return Type**: Boolean

**Examples**: If the "myEnum" attribute has the value "JOHN", then the Expression `${myEnum:in("PAUL", "JOHN", "MIKE")}` will return `true`. `${myEnum:in("RED", "GREEN", "BLUE")}` will return `false`.

### 1.1.7.5. find

**Description**: Returns `true` if the Subject contains any sequence of characters that matches the Regular Expression provided by the argument.

**Subject Type**: String

**Arguments**:

- *Regex* : The Regular Expression (in the Java Pattern syntax) to match against the Subject

**Return Type**: Boolean

**Examples**:

If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

#### Table 1.11. Table 11. find Examples

| Expression | Value |
|---|---|
| `${filename:find('a [Bb]rand [Nn]ew')}` | `true` |
| `${filename:find('Brand.*')}` | `false` |
| `${filename:find('brand')}` | `true` |

### 1.1.7.6. matches

**Description**: Returns `true` if the Subject exactly matches the Regular Expression provided by the argument.

**Subject Type**: String

**Arguments**:

- *Regex* : The Regular Expression (in the Java Pattern syntax) to match against the Subject

**Return Type**: Boolean

**Examples**:

If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

### Table 1.12. Table 12. matches Examples

| Expression | Value |
|---|---|
| `${filename:matches('a.*txt')}` | `true` |
| `${filename:matches('brand')}` | `false` |
| `${filename:matches('.brand.')}` | `true` |

## 1.1.7.7. indexOf

**Description**: Returns the index of the first character in the Subject that matches the String value provided as an argument. If the argument is found multiple times within the Subject, the value returned is the starting index of the **first** occurrence. If the argument cannot be found in the Subject, returns `-1`. The index is zero-based. This means that if the search string is found at the beginning of the Subject, the value returned will be `0`, not `1`.

**Subject Type**: String

**Arguments**:

- *value* : The value to search for in the Subject

**Return Type**: Number

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

### Table 1.13. Table 13. indexOf Examples

| Expression | Value |
|---|---|
| `${filename:indexOf('a.*txt')}` | `-1` |
| `${filename:indexOf('.')}` | `20` |
| `${filename:indexOf('a')}` | `0` |
| `${filename:indexOf('')}` | `1` |

## 1.1.7.8. lastIndexOf

**Description**: Returns the index of the first character in the Subject that matches the String value provided as an argument. If the argument is found multiple times within the Subject, the value returned is the starting index of the **last** occurrence. If the argument cannot be found in the Subject, returns `-1`. The index is zero-based. This means that if the search string is found at the beginning of the Subject, the value returned will be `0`, not `1`.

**Subject Type**: String

**Arguments**:

- *value* : The value to search for in the Subject

**Return Type**: Number

**Examples**: If the "filename" attribute has the value "a brand new filename.txt", then the following Expressions will provide the following results:

### Table 1.14. Table 14. lastIndexOf Examples

| Expression | Value |
|---|---|
| `${filename:lastIndexOf('a.*txt')}` | `-1` |
| `${filename:lastIndexOf('.')}` | `20` |
| `${filename:lastIndexOf('a')}` | `17` |
| `${filename:lastIndexOf('')}` | `11` |

## 1.1.7.9. jsonPath

**Description**: The `jsonPath` function generates a string by evaluating the Subject as JSON and applying a JSON path expression. An empty string is generated if the Subject does not contain valid JSON, the *jsonPath* is invalid, or the path does not exist in the Subject. If the evaluation results in a scalar value, the string representation of scalar value is generated. Otherwise a string representation of the JSON result is generated. A JSON array of length 1 is special cased when `[0]` is a scalar, the string representation of `[0]` is generated.[1]

**Subject Type**: String

**Arguments**: *jsonPath* : the JSON path expression used to evaluate the Subject.

**Return Type**: String

**Examples**: If the "myJson" attribute is

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

**Table 1.15. Table 15. jsonPath Examples**

| Expression | Value |
|---|---|
| `${myJson:jsonPath('$.firstName')}` | `John` |
| `${myJson:jsonPath('$.address.postalCode')}` | `10021-3100` |
| `${myJson:jsonPath('$.phoneNumbers[?(@.type=="home")].number')}`[1] | `212 555-1234` |
| `${myJson:jsonPath('$.phoneNumbers')}` | `[{"type":"home","number":"212 555-1234"},{"type":"office","number":"646 555-4567"}]` |
| `${myJson:jsonPath('$.missing-path')}` | *empty* |
| `${myJson:jsonPath('$.bad-json-path..')}` | *exception bulletin* |

An empty subject value or a subject value with an invalid JSON document results in an exception bulletin.

## 1.1.8. Mathematical Operations and Numeric Manipulation

For those functions that support Decimal and Number (whole number) types, the return value type depends on the input types. If either the subject or argument are a Decimal then the result will be a Decimal. If both values are Numbers then the result will be a Number. This includes Divide. This is to preserve backwards compatibility and to not force rounding errors.

### 1.1.8.1. plus

**Description**: Adds a numeric value to the Subject. If either the argument or the Subject cannot be coerced into a Number, returns `null`.

**Subject Type**: Number or Decimal

**Arguments**:

- *Operand* : The value to add to the Subject

**Return Type**: Number or Decimal (depending on input types)

**Examples**: If the "fileSize" attribute has a value of 100, then the Expression `${fileSize:plus(1000)}` will return the value `1100`.

### 1.1.8.2. minus

**Description**: Subtracts a numeric value from the Subject.

**Subject Type**: Number or Decimal

**Arguments**:

- *Operand* : The value to subtract from the Subject

**Return Type**: Number or Decimal (depending on input types)

**Examples**: If the "fileSize" attribute has a value of 100, then the Expression `${fileSize:minus(100)}` will return the value `0`.

#### 1.1.8.3. multiply

**Description**: Multiplies a numeric value by the Subject and returns the product.

**Subject Type**: Number or Decimal

**Arguments**:

- *Operand* : The value to multiple the Subject by

**Return Type**: Number or Decimal (depending on input types)

**Examples**: If the "fileSize" attribute has a value of 100, then the Expression `${fileSize:multiply(1024)}` will return the value `102400`.

#### 1.1.8.4. divide

**Description**: Divides the Subject by a numeric value and returns the result.

**Subject Type**: Number or Decimal

**Arguments**:

- *Operand* : The value to divide the Subject by

**Return Type**: Number or Decimal (depending on input types)

**Examples**: If the "fileSize" attribute has a value of 100, then the Expression `${fileSize:divide(12)}` will return the value `8`.

#### 1.1.8.5. mod

**Description**: Performs a modular division of the Subject by the argument. That is, this function will divide the Subject by the value of the argument and return not the quotient but rather the remainder.

**Subject Type**: Number or Decimal

**Arguments**:

- *Operand* : The value to divide the Subject by

**Return Type**: Number or Decimal (depending on input types)

**Examples**: If the "fileSize" attribute has a value of 100, then the Expression `${fileSize:mod(12)}` will return the value `4`.

#### 1.1.8.6. toRadix

**Description**: [.description]#Converts the Subject from a Base 10 number to a different Radix (or number base). An optional second argument can be used to indicate the minimum number of characters to be used. If the converted value has fewer than this number of characters, the number will be padded with leading zeroes.

```
If a decimal is passed as the subject, it will first be converted to a whole
 number and then processed.#
```

**Subject Type**: Number

**Arguments**:

- *Desired Base* : A Number between 2 and 36 (inclusive)
- *Padding* : Optional argument that specifies the minimum number of characters in the converted output

**Return Type**: String

**Examples**: If the "fileSize" attributes has a value of 1024, then the following Expressions will yield the following results:

### Table 1.16. Table 16. toRadix Examples

| Expression | Value |
|---|---|
| `${fileSize:toRadix(10)}` | `1024` |
| `${fileSize:toRadix(10, 1)}` | `1024` |
| `${fileSize:toRadix(10, 8)}` | `00001024` |
| `${fileSize:toRadix(16)}` | `400` |
| `${fileSize:toRadix(16, 8)}` | `00000400` |
| `${fileSize:toRadix(2)}` | `10000000000` |
| `${fileSize:toRadix(2, 16)}` | `0000010000000000` |

## 1.1.8.7. fromRadix

**Description**: [.description]#Converts the Subject from a specified Radix (or number base) to a base ten whole number. The subject will converted as is, without interpretation, and all characters must be valid for the base being converted from. For example converting "0xFF" from hex will not work due to "x" being a invalid hex character.

```
If a decimal is passed as the subject, it will first be converted to a whole
 number and then processed.#
```

**Subject Type**: String

**Arguments**:

- *Subject Base* : A Number between 2 and 36 (inclusive)

**Return Type**: Number

**Examples**: If the "fileSize" attributes has a value of 1234A, then the following Expressions will yield the following results:

### Table 1.17. Table 17. toRadix Examples

| Expression | Value |
|---|---|
| `${fileSize:fromRadix(11)}` | `17720` |
| `${fileSize:fromRadix(16)}` | `74570` |

| | |
|---|---|
| `${fileSize:fromRadix(20)}` | `177290` |

### 1.1.8.8. random

**Description**: Returns a random whole number ( 0 to 2^63 - 1) using an insecure random number generator.

**Subject Type**: No subject

**Arguments**: No arguments

**Return Type**: Number

**Examples**: ${random():mod(10):plus(1)} returns random number between 1 and 10 inclusive.

### 1.1.8.9. math

**Description**: ADVANCED FEATURE. This expression is designed to be used by advanced users only. It utilizes Java Reflection to run arbitrary java.lang.Math static methods. The exact API will depend on the version of Java you are running. The Java 8 API can be found here: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

In order to run the correct method, the parameter types must be correct. The Expression Language "Number" (whole number) type is interpreted as a Java "long". The "Decimal" type is interpreted as a Java "double". Running the desired method may require calling "toNumber()" or "toDecimal()" in order to "cast" the value to the desired type. This also is important to remember when cascading "math()" calls since the return type depends on the method that was run.

**Subject Type**: Subjectless, Number or Decimal (depending on the desired method to run)

**Arguments**: - *Method* : The name of the Java Math method to run - *Optional Argument* : Optional argument that acts as the second parameter to the method.

**Return Type**: Number or Decimal (depending on method run)

**Examples**:

- ${math("random")} runs Math.random().
- ${literal(2):toDecimal:math("pow", 2.5)} runs Math.pow(2D,2.5D).
- ${literal(64):toDouble():math("cbrt"):toNumber():math("max", 5)} runs Math.maxDouble.valueOf(Math.cbrt(64D).longValue(), 5L). Note that the toDecimal() is needed because "cbrt" takes a "double" as input and the "64" will get interpreted as a long. The "toDecimal()" call is necessary to correctly call the method. that the "toNumber()" call is necessary because "cbrt" returns a double and the "max" method is must have parameters of the same type and "5" is interpreted as a long.
- ${literal(5.4):math("scalb", 2)} runs Math.scalb(5.4, 2). This example is important because NiFi EL treats all whole numbers as "longs" and there is no concept of an "int". "scalb" takes a second parameter of an "int" and it is not overloaded to accept longs so it could not be run without special type handling. In the instance where the Java method cannot

be found using parameters of type "double" and "long" the "math()" EL function will attempt to find a Java method with the same name but parameters of "double" and "int".

- ${first:toDecimal():math("pow", ${second:toDecimal()})} where attributes evaluate to "first" = 2.5 and "second" = 2. This example runs Math.pow(2.5D, 2D). The explicit calls to toDecimal() are important because of the dynamic nature of EL. When creating the flow, the user is unaware if the expression language values will be able to be interpreted as a whole number or not. In this example without the explicit calls "toDecimal" the "math" function would attempt to run a Java method "pow" with types "double" and "long" (which doesn't exist).

# 1.1.9. Date Manipulation

## 1.1.9.1. format

**Description**: Formats a number as a date/time according to the format specified by the argument. The argument must be a String that is a valid Java SimpleDateFormat format. The Subject is expected to be a Number that represents the number of milliseconds since Midnight GMT on January 1, 1970. The number will be evaluated using the local time zone unless specified in the second optional argument.

**Subject Type**: Number

**Arguments**:

- *format* : The format to use in the Java SimpleDateFormat syntax
- *time zone* : Optional argument that specifies the time zone to use (in the Java TimeZone syntax)

**Return Type**: String

**Examples**: If the attribute "time" has the value "1420058163264", then the following Expressions will yield the following results:

### Table 1.18. Table 18. format Examples

| Expression | Value |
|---|---|
| `${time:format("yyyy/MM/dd HH:mm:ss.SSS'Z'", "GMT")}` | `2014/12/31 20:36:03.264Z` |
| `${time:format("yyyy/MM/dd HH:mm:ss.SSS'Z'", "America/Los_Angeles")}` | `2014/12/31 12:36:03.264Z` |
| `${time:format("yyyy/MM/dd HH:mm:ss.SSS'Z'", "Asia/Tokyo")}` | `2015/01/01 05:36:03.264Z` |
| `${time:format("yyyy/MM/dd", "GMT")}` | `2014/12/31` |
| `${time:format("HH:mm:ss.SSS'Z'", "GMT")}` | `20:36:03.264Z` |
| `${time:format("yyyy", "GMT")}` | `2014` |

## 1.1.9.2. toDate

**Description**: Converts a String into a Date data type, based on the format specified by the argument. The argument must be a String that is a valid Java SimpleDateFormat syntax. The Subject is expected to be a String that is formatted according the argument. The

date will be evaluated using the local time zone unless specified in the second optional argument.

**Subject Type**: String

**Arguments**:

- *format* : The current format to use when parsing the Subject, in the Java SimpleDateFormat syntax.
- *time zone* : Optional argument that specifies the time zone to use when parsing the Subject, in the Java TimeZone syntax.

**Return Type**: Date

**Examples**: If the attribute "year" has the value "2014" and the attribute "time" has the value "2014/12/31 15:36:03.264Z", then the Expression `${year:toDate('yyyy', 'GMT')}` will return a Date data type with a value representing Midnight GMT on January 1, 2014. The Expression `${time:toDate("yyyy/MM/dd HH:mm:ss.SSS'Z'", "GMT")}` will result in a Date data type for 15:36:03.264 GMT on December 31, 2014.

Often, this function is used in conjunction with the format function to change the format of a date/time. For example, if the attribute "date" has the value "12-24-2014" and we want to change the format to "2014/12/24", we can do so by chaining together the two functions: `${date:toDate('MM-dd-yyyy'):format('yyyy/MM/dd')}`.

### 1.1.9.3. now

**Description**: Returns the current date and time as a Date data type object.

**Subject Type**: No Subject

**Arguments**: No arguments

**Return Type**: Date

**Examples**: We can get the current date and time as a Date data type by using the `now` function: `${now()}`. As an example, on Wednesday December 31st 2014 at 36 minutes after 3pm and 36.123 seconds EST `${now()}` would be evaluated to be a Date type representing that time. Since whole Expression Language expressions can only return Strings it would formatted as `Wed Dec 31 15:36:03 EST 2014` when the expression completes.

Utilizing the toNumber method, `now` can provide the current date and time as the number of milliseconds since Midnight GMT on January 1, 1970. For instance, if instead of executing `${now()}` in the previous example `${now():toNumber()}` was run then it would output `1453843201123`. This method provides millisecond-level precision and provides the ability to manipulate the value.

#### Table 1.19. Table 19. now Examples

| Expression | Value |
|---|---|
| `${now()}` | A Date type representing the current date and time to the nearest millisecond |

| | |
|---|---|
| `${now():toNumber()}` | The number of milliseconds since midnight GMT Jan 1, 1970 (1453843201123, for example) |
| `${now():toNumber():minus(86400000)}` | A number presenting the time 24 hours ago |
| `${now():format('yyyy')}` | The current year |
| `${now():toNumber():minus(86400000):format('EEE')}` | The day of the week that was yesterday, as a 3-letter abbreviation (For example, `Wed`) |

# 1.1.10. Type Coercion

## 1.1.10.1. toString

**Description**: Coerces the Subject into a String

**Subject Type**: Any type

**Arguments**: No arguments

**Return Type**: String

**Examples**: The Expression `${fileSize:toNumber():toString()}` converts the value of "fileSize" attribute to a number and back to a String.

## 1.1.10.2. toNumber

**Description**: Coerces the Subject into a Number

**Subject Type**: String, Decimal, or Date

**Arguments**: No arguments

**Return Type**: Number

**Examples**: The Expression `${fileSize:toNumber()}` converts the attribute value of "fileSize" to a number.

## 1.1.10.3. toDecimal

**Description**: Coerces the Subject into a Decimal

**Subject Type**: String, Whole Number or Date

**Arguments**: No arguments

**Return Type**: Decimal

**Examples**: The Expression `${fileSize:toDecimal()}` converts the attribute value of "fileSize" to a decimal.

# 1.1.11. Subjectless Functions

While the majority of functions in the Expression Language are called by using the syntax `${attributeName:function()}`, there exist a few functions that are not expected to have subjects. In this case, the attribute name is not present. For example, the IP address of the machine can be obtained by using the Expression `${ip()}`. All of the functions in

this section are to be called without a subject. Attempting to call a subjectless function and provide it a subject will result in an error when validating the function.

#### 1.1.11.1. ip

**Description**: Returns the IP address of the machine.

**Subject Type**: No subject

**Arguments**: No arguments

**Return Type**: String

**Examples**: The IP address of the machine can be obtained by using the Expression `${ip()}`.

#### 1.1.11.2. hostname

**Description**: Returns the Hostname of the machine. An optional argument of type Boolean can be provided to specify whether or not the Fully Qualified Domain Name should be used. If `false`, or not specified, the hostname will not be fully qualified. If the argument is `true` but the fully qualified hostname cannot be resolved, the simple hostname will be returned.

**Subject Type**: No subject

**Arguments**:

- *Fully Qualified* : Optional parameter that specifies whether or not the hostname should be fully qualified. If not specified, defaults to false.

**Return Type**: String

**Examples**: The fully qualified hostname of the machine can be obtained by using the Expression `${hostname(true)}`, while the simple hostname can be obtained by using either `${hostname(false)}` or simply `${hostname()}`.

#### 1.1.11.3. UUID

**Description**: Returns a randomly generated UUID.

**Subject Type**: No Subject

**Arguments**: No arguments

**Return Type**: String

**Examples**: ${UUID()} returns a value similar to de305d54-75b4-431b-adb2-eb6b9e546013

#### 1.1.11.4. nextInt

**Description**: Returns a one-up value (starting at 0) and increasing over the lifetime of the running instance of NiFi. This value is not persisted across restarts and is not guaranteed to be unique across a cluster. This value is considered "one-up" in that if called multiple

times across the NiFi instance, the values will be sequential. However, this counter is shared across all NiFi components, so calling this function multiple times from one Processor will not guarantee sequential values within the context of a particular Processor.

**Subject Type**: No Subject

**Arguments**: No arguments

**Return Type**: Number

**Examples**: If the previous value returned by `nextInt` was 5, the Expression `${nextInt():divide(2)}` obtains the next available integer (6) and divides the result by 2, returning a value of 3.

### 1.1.11.5. literal

**Description**: Returns its argument as a literal String value. This is useful in order to treat a string or a number at the beginning of an Expression as an actual value, rather than treating it as an attribute name. Additionally, it can be used when the argument is an embedded Expression that we would then like to evaluate additional functions against.

**Subject Type**: No Subject

**Arguments**:

- *value* : The value to be treated as a literal string, number, or boolean value.

**Return Type**: String

**Examples**: `${literal(2):gt(1)}` returns `true`

`${literal( ${allMatchingAttributes('a.*'):count()} ):gt(3)}` returns true if there are more than 3 attributes whose names begin with the letter `a`.

### 1.1.11.6. getStateValue

**Description**: Access a processor's state values by passing in the String key and getting the value back as a String. This is a special Expression Language function that only works with processors that explicitly allow EL to query state. Currently only UpdateAttribute does.

**Subject Type**: No Subject

**Arguments**:

- *Key* : The key to use when accessing the state map.

**Return Type**: String

**Examples**: UpdateAttribute processor has stored the key "count" with value "20" in state. '${getStateValue("count")}` returns `20`.

## 1.1.12. Evaluating Multiple Attributes

When it becomes necessary to evaluate the same conditions against multiple attributes, this can be accomplished by means of the `and` and `or` functions. However, this quickly

becomes tedious, error-prone, and difficult to maintain. For this reason, NiFi provides several functions for evaluating the same conditions against groups of attributes at the same time.

### 1.1.12.1. anyAttribute

**Description**: Checks to see if any of the given attributes, match the given condition. This function has no subject and takes one or more arguments that are the names of attributes to which the remainder of the Expression is to be applied. If any of the attributes specified, when evaluated against the rest of the Expression, returns a value of `true`, then this function will return `true`. Otherwise, this function will return `false`.

**Subject Type**: No Subject

**Arguments**:

- *Attribute Names* : One or more attribute names to evaluate

**Return Type**: Boolean

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "filename" contains "file.txt" consider the following examples:

#### Table 1.20. Table 20. anyAttribute Examples

| Expression | Value |
|---|---|
| `${anyAttribute("abc", "xyz"):contains("bye")}` | `true` |
| `${anyAttribute("filename","xyz"):toUpper():contains("e")}` | `false` |

### 1.1.12.2. allAttributes

**Description**: Checks to see if all of the given attributes match the given condition. This function has no subject and takes one or more arguments that are the names of attributes to which the remainder of the Expression is to be applied. If all of the attributes specified, when evaluated against the rest of the Expression, returns a value of `true`, then this function will return `true`. Otherwise, this function will return `false`.

**Subject Type**: No Subject

**Arguments**:

- *Attribute Names* : One or more attribute names to evaluate

**Return Type**: Boolean

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "filename" contains "file.txt" consider the following examples:

#### Table 1.21. Table 21. allAttributes Example

| Expression | Value |
|---|---|
| `${allAttributes("abc", "xyz"):contains("world")}` | `true` |

| | |
|---|---|
| `${allAttributes("abc", "filename","xyz"):toUpper():contains("e")}` | `false` |

### 1.1.12.3. anyMatchingAttribute

**Description**: Checks to see if any of the given attributes, match the given condition. This function has no subject and takes one or more arguments that are Regular Expressions to match against attribute names. Any attribute whose name matches one of the supplied Regular Expressions will be evaluated against the rest of the Expression. If any of the attributes specified, when evaluated against the rest of the Expression, returns a value of `true`, then this function will return `true`. Otherwise, this function will return `false`.

**Subject Type**: No Subject

**Arguments**:

- *Regex* : One or more Regular Expressions (in the Java Pattern syntax) to evaluate against attribute names

**Return Type**: Boolean

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "filename" contains "file.txt" consider the following examples:

#### Table 1.22. Table 22. anyMatchingAttribute Example

| Expression | Value |
|---|---|
| `${anyMatchingAttribute("[ax].*"):contains('bye')}` | `true` |
| `${anyMatchingAttribute(".*"):isNull()}` | `false` |

### 1.1.12.4. allMatchingAttributes

**Description**: Checks to see if any of the given attributes, match the given condition. This function has no subject and takes one or more arguments that are Regular Expressions to match against attribute names. Any attribute whose name matches one of the supplied Regular Expressions will be evaluated against the rest of the Expression. If all of the attributes specified, when evaluated against the rest of the Expression, return a value of `true`, then this function will return `true`. Otherwise, this function will return `false`.

**Subject Type**: No Subject

- *Regex* : One or more Regular Expressions (in the Java Pattern syntax) to evaluate against attribute names

**Return Type**: Boolean

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "filename" contains "file.txt" consider the following examples:

#### Table 1.23. Table 23. anyMatchingAttributes Examples

| Expression | Value |
|---|---|
| `${allMatchingAttributes("[ax].*"):contains("world")}` | `true` |

| | |
|---|---|
| `${allMatchingAttributes(".*"):isNull()}` | `false` |
| `${allMatchingAttributes("f.*"):count()}` | `1` |

### 1.1.12.5. anyDelineatedValue

**Description**: Splits a String apart according to a delimiter that is provided, and then evaluates each of the values against the rest of the Expression. If the Expression, when evaluated against any of the individual values, returns `true`, this function returns `true`. Otherwise, the function returns `false`.

**Subject Type**: No Subject

**Arguments**:

- *Delineated Value* : The value that is delineated. This is generally an embedded Expression, though it does not have to be.
- *Delimiter* : The value to use to split apart the *delineatedValue* argument.

**Return Type**: Boolean

**Examples**: Given that the "number_list" attribute contains the value "1,2,3,4,5", and the "word_list" attribute contains the value "the,and,or,not", consider the following examples:

#### Table 1.24. Table 24. anyDelineatedValue Examples

| Expression | Value |
|---|---|
| `${anyDelineatedValue("${number_list}", ","):contains("5")}` | `true` |
| `${anyDelineatedValue("this that and", ","):equals("${word_list}")}` | `false` |

### 1.1.12.6. allDelineatedValues

**Description**: Splits a String apart according to a delimiter that is provided, and then evaluates each of the values against the rest of the Expression. If the Expression, when evaluated against all of the individual values, returns `true` in each case, then this function returns `true`. Otherwise, the function returns `false`.

**Subject Type**: No Subject

**Arguments**:

- *Delineated Value* : The value that is delineated. This is generally an embedded Expression, though it does not have to be.
- *Delimiter* : The value to use to split apart the *delineatedValue* argument.

**Return Type**: Boolean

**Examples**: Given that the "number_list" attribute contains the value "1,2,3,4,5", and the "word_list" attribute contains the value "those,known,or,not", consider the following examples:

### Table 1.25. Table 25. allDelineatedValues Examples

| Expression | Value |
|---|---|
| `${allDelineatedValues("${word_list}", ","):contains("o")}` | `true` |
| `${allDelineatedValues("${number_list}", ","):count()}` | `4` |
| `${allDelineatedValues("${number_list}", ","):matches("[0-9]+")}` | `true` |
| `${allDelineatedValues("${word_list}", ","):matches('e')}` | `false` |

## 1.1.12.7. join

**Description**: Aggregate function that concatenates multiple values with the specified delimiter. This function may be used only in conjunction with the `allAttributes`, `allMatchingAttributes`, and `allDelineatedValues` functions.

**Subject Type**: String

**Arguments**:

- *Delimiter* : The String delimiter to use when joining values

**Return Type**: String

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "filename" contains "file.txt" consider the following examples:

### Table 1.26. Table 26. join Examples

| Expression | Value |
|---|---|
| `${allMatchingAttributes("[ax].*"):substringBefore("d"):join("-")}` | `hello worl-good` |
| `${allAttributes("abc", "xyz"):join(" now")}` | `hello world nowgood bye world now` |

## 1.1.12.8. count

**Description**: Aggregate function that counts the number of non-null, non-false values returned by the `allAttributes`, `allMatchingAttributes`, and `allDelineatedValues`. This function may be used only in conjunction with the `allAttributes`, `allMatchingAttributes`, and `allDelineatedValues` functions.

**Subject Type**: Any

**Arguments**: No arguments

**Return Type**: Number

**Examples**: Given that the "abc" attribute contains the value "hello world", "xyz" contains "good bye world", and "number_list" contains "1,2,3,4,5" consider the following examples:

### Table 1.27. Table 27. count Examples

| Expression | Value |
|---|---|

| | |
|---|---|
| ${allMatchingAttributes("[ax].*"):substringBefore(""):count()} | 2 |
| ${allAttributes("abc", "xyz"):contains("world"):count()} | 1 |
| ${allDelineatedValues(${number_list}, ","):count()} | 5 |
| ${allAttributes("abc", "non-existent-attr", "xyz"):count()} | 2 |
| ${allMatchingAttributes(".*"):length():gt(10):count()} | 2 |