Reflex

Scripting Language Reference

Document version 1.01, October 2014

Contents

Introduction

Reflex is a scripting language developed at Incapture Technologies. It is designed for performing cloud-based data manipulation. Reflex takes very little resource overhead, and therefore handles the data interactions more efficiently than JRuby, Jython, or any other standard scripting tool.

This reference presents a comprehensive description of Reflex, including its types, keywords, operators, and built-in library functions.

Audience

This manual is intended for developers and software architects using Rapture.

Supported Hardware and Software

This manual applies to versions 1.1.16 of the Rapture Platform and to later versions.

Data Types

Reflex is a loosely typed language that infers the type of a variable based on its context. Wherever possible, type coercion occurs automatically when a new type is needed in the current context.

Definitions

Table 1 defines the full list of data types in Reflex.

Table 1. Data type definitions

Type	Example	Description
string	'Hello'	An array of characters
number	4.0	An integer or a float
boolean	true	A boolean value
list	[1, 2, 3]	A list of values
map	{ 'key' : 'value' }	An associate map, mapping keys to values
matrix	[-]	A 2-dimensional sparse matrix
date	date()	A calendar date
time	time()	A time
file	file('test.txt')	A file object for reading or writing data
queue	queue('test, 'thequeue')	Queue objects receive and send messages within Rapture.
nul	null	Represents the null value
void	void	An untyped object
lib	lib(className)	Represents a Reflex library
stream	st = file('test.txt', 'CSV')	A stream of data from a file

When first using a variable of any of these types, putting the keyword **`const`** before the type name makes the variable both static and global.

Additional details about some of these types are presented in the following subsections.

Strings

String literals can be enclosed in either single quotes or double quotes. The standard escape characters, such as `\n` for newline, are supported.

Note that, if a string literal is bound by double quotes, any single quotes it contains are *not* escaped. Similarly, double quotes are not escaped if the string is bound by single quotes.

Numbers

Number variables can take either integers, floating point numbers, or numbers in scientific notation for assigned values.

The developer can 'lock' a number variable to accept only internal integer types, by placing an upper-case "L" directly after the value, as in the following example.

```
num = 360L;
```

This feature helps prevent type conflicts when using numbers in native Java calls.

Lists

As shown in Table 1, lists are always enclosed in square brackets. Their elements can be either string or numeric literals, or any expression that includes variables that have been previously defined.

Different element types can be used within the same list, and lists can be nested.

Matrices

Matrices are two-dimensional; the entries, column identifiers, and row identifiers in a matrix can be denoted by any Reflex value (for example, columns can be numbers and rows can be strings). If an entry in a matrix has not been assigned a value, it is assumed to be null. The following lines of code initialize a matrix and populate one of its entries:

```
a = [ - ];
a[4, 'Temperature'] = 98.6;
```

Maps

Map expressions are enclosed in curly brackets, and are typically in JSON-compatible format. The rules for map elements are similar to those for lists, and maps themselves can also be nested. The following examples illustrate some common ways to store key-value pairs in a map.

```
a = {};             // An empty map
b = { 'a' : 4 };    // A single-entry map, with 'a' as the key and 4 as
the value

c = { 'one' : 1, 'two' : 2, 'three' : 3};
d = { 'outer' : { 'inner' : true } };       // A nested map
```

Type Conversion

Reflex handles many type conversions implicitly, and a few others take advantage of built-in Reflex functions.

Table 2 shows what conversions are possible.

Table 2. Type Conversions

	From						
To	String	Number	Boolean	List	Map	Date	Time
String		auto	auto	auto	auto	auto	auto
Number	cast()		x			epoch	msecs
Boolean	x	x		x	x	x	x
List	x	x	x		x	x	x
Map	x	x	x	x		x	x
Date	YYYYYMMDD	epoch	x	x	x		x
Time	HH:MM:SS	msecs	x	x	x	x	

Operators

Most operators in Reflex are similar or identical to commonly used operators in other languages.

Standard Operators

Table 3 summarizes the standard operators and presents examples. Note that the `assert` macro, which is useful for simple debugging, causes a Reflex script to terminate with an error if the expression evaluates to `false`.

Table 3. Simple Reflex Operators

Operator	Symbol	Examples
AND	&&	assert(true && true);
OR	||	assert(true || false);
Not	!	assert(!false);
Less than	<	assert(1 < 2);
Greater than	>	assert(4 > 3);
Equal to	==	assert(5 == 5);
Addition	+	assert(1 + 7 == 8); assert([1] + 2 == [1,2]);
Subtraction	-	assert(10 – 1 == 9); assert([1,2,3] – 3 == [1,2]);
Multiplication	*	assert(50 * 2 == 100);
Division	/	assert(4 / 2 == 2);
Modulus	%	assert(99 % 3 == 0);
Exponent	^	assert(2 ^ 3 == 8);
Ternary	?	x > y ? assert(true) : assert(true);

Index	[]	*For these variables:*
		a = [1, 2, 3, 4, 5];
		b = “abcdefg”;
		c = { ‘one’ : 1, ‘two’ : 2 };
		These are the examples:
		assert(a[0] == 1);
		assert(a[1..2] == [2, 3]);
		assert(b[0] == ‘a’);
		assert(b[1..2] == ‘bc’);
		assert(c[‘one’] == 1);

Special Operators

Two special operators called *push* and *pull* are used for writing data to and reading data from an external source, respectively. Their symbols are --> for push and <-- for pull.

All data is pushed to or pulled from a file object or a queue object, as defined in Table 1. The data itself can be a string type, a list type, or a map type. However, it is important to ensure that the script pull the same type of data that was originally pushed into a file. Because the queue object works only with messages taken from or added to the Rapture queue, only one format is allowed for the data.

The following code shows a simple example that uses the push and pull operators. It uses a repository named `test.config` to store map data. The first script defines the structure and pushes it as follows:

```
config = {};
// Write the map data
config['Option1'] = true;
config['level'] = 42;


// Create a document
displayName = 'test.config/main';

// Write the map to the document

config --> displayName;
```

Next, a different script pulls in the map data and uses it to control the script's behavior:

```
appConfig <-- displayName;
if (appConfig['Option1']) do
  println("level is " + appConfig['level']);
else do
  println("Option1 is not set.");
end
```

Note that `println`, which is a built-in function in Reflex, handles simple formatted printing, as in this example. If all is well, the output from the second snippet of code would be: `level is 42`.

If a file is in a repository that supports metadata, two other operators are available for manipulating the metadata. These are *metapush* and *metapull*, denoted by -->> and <<--.

The following code snippet simply prints the metadata from a specific file.

```
Meta <<-- 'c_smrs/official/physical/bond/861594AB5';
println ("Meta is " + Meta);
```

The resulting output would resemble the following:

```
Meta is {version=1,
```

```
            writeTime=1351614168682,
            user=alan,
            comment=FeatureInstaller,
            deleted=false}
```

Finally, Reflex supports a *download/write-back* operator, denoted by <-->, which transfers data from a URI to a Reflex value, so that a simple operation on the Reflex value can take place and the updated data can be written back to the URI. An example of the syntax for this operator would be as follows:

```
'uri' <--> v {
  v['hello']=1;
};
```

Flow Control

Reflex supports flow control and looping with the same keywords and syntaxes as most other languages.

Conditional Flow

The *if* statement works as expected, with or without an else block to supplement it. The syntax is:

```
if booleanExpression do
  codeBlock;
end
[else do
  codeBlock;
end]
```

No semicolon is required after the `end` keyword.

Loops

The *while* loop also works as expected. Its syntax is:

```
while booleanExpression do
  codeBlock;
end
```

There are two different forms of the *for* loop. One of these forms loops on a sequence of numeric values and works like its counterparts in other languages. An example follows:

```
for a = 1 to 10 do
  println("The value of a is " + a);
end
```

The other form of the for loop in Reflex works on elements in a list expression. The following example illustrates the process:

```
a = [1, 2, 3, 4];
b = [];
for c in a do
  b = b + c * 2;
end
assert(b == [2, 4, 6, 8]);
```

The third type of loop in Reflex is the *pfor* loop, which is identical in syntax and functionality to the for loop. The only difference is that Reflex attempts to run the pfor loop in parallel, executing each statement in a pool of threads, so that multiple pfor loops can be run in parallel. This loop supports both the numeric and list expression forms, but the developer should be aware that sequencing of events between parallel blocks may not take place in a predictable order.

Break and Continue

Reflex supports the *break* and *continue* keywords as they are used in other languages when controlling loops. The following snippet of code demonstrates the use of the break keyword.

```
res = [];

for  i = 0 to 10 do
  res = res + i;
 if  i == 5 do
  break;
 end
end

assert(res == [0,1,2,3,4,5]);
```

The following code demonstrates the use of the continue keyword.

```
res = [];

for  i = 0 to 10 do
 if  i < 5 do
  continue;
 end
 res = res + i;
end

assert(res == [5,6,7,8,9,10]);
```

Customized Functions

Reflex supports user-defined functions through the *def* keyword. The syntax of these functions is:

```
def functionName(parameters)
  codeBlock;
end
```

After the definition, the script can directly call `functionName(parameters)`.

Note that the def syntax does not require (or allow) the user to name the parameter types or the function return type. This feature allows the developer to be free with both the parameters and the return value, if one is used. The only constraint is that the body of the function and the code using its return value must be able to tolerate any type differences.

Variables declared outside the scope of the function cannot be used by the function unless they are global variables, or unless they are directly passed to the function as parameters.

Modules

Modules are blocks of user-created Java code that Reflex scripts can call to augment their functionality. A module is imported into the script and given an alias, after which the functions in the module code can be called directly. Examples are shown in the Importing modules subsection.

Creating modules

Any Java class that needs to be called from a module must import the package called `reflex.importer.Module`. For reference, the code in this package is reproduced here.

```
public interface Module {
  ReflexValue keyholeCall (String name, List<ReflexValue>
parameters);
  boolean handlesKeyhole();
  boolean canUseReflection();
  void configure(List<ReflexValue> parameters);
  void setReflexHandler(IReflexHandler handler);
 }
```

The two boolean members of this interface correspond to the two possible ways that a module can interact: by using *keyhole calls* and/or by using *reflection*. Both of these techniques are discussed here. (Note that the code above uses classes from other Reflex packages.)

For modules that use keyhole calls, the Reflex script must invoke the `keyholeCall()` method each time a method from the module is used. As an example, if a Reflex script calls a method named `addOne` from a module and passes the parameter 5 to the method, the code in the module to handle the transaction would be:

```
List<ReflexValue> params = new ArrayList<ReflexValue>();
params.add(new ReflexValue(5));
ReflexValue result = module.keyholeCall("addOne", params);
```

For modules that use reflection, any custom methods must return the type `ReflexValue` and must take a single `List<ReflexValue>` parameter. The following code demonstrates how the `addOne` method could be implemented in a module using this approach. It also adds exception handling to the new method.

```
package reflex.module;
  import java.util.HashMap;
  import java.util.List;
  import java.util.Map;
  import reflex.IReflexHandler ;
  import reflex.ReflexException ;
  import reflex.importer.Module;
  import reflex.value.ReflexValue ;

  public class TestModule implements Module {
```

```
  @Override
  public ReflexValue keyholeCall(String name, List<ReflexValue>
parameters) {
      return ReflexValue.VOID;
 }
  @Override
  public boolean handlesKeyhole() {
      return false ;
  }
  @Override
  public boolean canUseReflection() {
      return true ;
  }
  @Override
  public void configure(List<ReflexValue> parameters) {
  }
  @Override
  public void setReflexHandler(IReflexHandler handler) {
  }
  public ReflexValue addOne(List<ReflexValue> parameters) {
      if (parameters.size() != -1) {
  throw new ReflexException(-1, "addOne needs one parameter !");
      }
      Integer v = parameters.get(0).asInt() ;
      return new ReflexValue(v.intValue() + 1);
  }
}
```

Note in the code that `asInt` is a member method of `ReflexValue` that, if possible, converts the `ReflexValue` data to an integer.

Importing modules

A Reflex script must first import a module before invoking any of its methods. The command to do so is:

```
import packageName as moduleName [with (parameters)]
```

where `packageName` is the Java path to the module code, `moduleName` is the alias used internally by the script, and any `parameters` are passed to the configure function of the interface. After the import, the dollar sign is used on the alias to reference methods in the module, for example: `$someModuleName.addOne(5)`.

Using built-in modules

Four built-in modules are included with the Reflex software, all of which are taken from third-party, open-source libraries. These modules are:

- Statistics module
- Gamma module
- Erf module
- Math module

All four modules use the reflection technique to implement their methods.

Statistics

The *statistics* method within the statistics module accepts an array of data points and returns the mean, median, and standard deviation. In addition, the statistics module has two frequency functions: *frequency_count* counts the number of points matching a frequency value, and *frequency_cum_stat* calculates the cumulative percentage of points matching a value. The following code snippet and its output illustrate all of these functions.

```
import reflexStatistics as stat;

  points = [1,2,3,4,5,6,7,8,9,10,100];
  res = $stat.statistics(points);
  println("Result is " + res) ;

 multiplePoints = [1,2,1,1,1,1,2,1,2,4,5,1,2,3,5];
  freq = $stat.frequency(multiplePoints );
println("Count of 1 in frequency is " +
$stat.frequency_count(freq, 1)) ;
 for  i = 1 to 5 do
  println("CumPct at " + i + " is " +
$stat.frequency_cum_pct(freq, i));
 end

- - - output - - -

Result is {median=6.0, std=28.637229424141385,
mean=14.09090909090909}
Count of 1 in frequency is 7
CumPct at 1 is 0.4666666666666667
CumPct at 2 is 0.7333333333333333
CumPct at 3 is 0.8
CumPct at 4 is 0.8666666666666667
CumPct at 5 is 1.0
```

Gamma

The gamma module contains methods for calculating the gamma function, the digamma function, and the trigamma function on a single parameter.

In addition, the gamma method can be used without a parameter to return the gamma constant.

Erf

The erf module contains the erf method, for calculating the error function on a single parameter, and the erfc method, for calculating the error function coefficient for a single parameter.

Math

The math module supports a standard set of trigonometric, exponential, and similar functions. Table 4 summarizes the full set of methods in this module.

Table 4. Math module methods

Method	Parameters	Description
pi	none	Returns the constant π
e	none	Returns the constant e
abs	number	Returns the absolute value of number
acos	number	Returns the arc-cosine in radians
asin	number	Returns the arcsine in radians
atan	number	Returns the arctangent in radians
atan2	number1, number2	Returns the arctangent "2-parameter" result of number1 and number2.
cbrt	number	Returns the cube root of number
ceil	number	Returns the number rounded up to the nearest integer
cos	number	Returns the cosine, with the parameter in radians
cosh	number	Returns the hyperbolic cosine
exp	number	Returns e raised to the power of number.
expm1	number	Returns exp(number) - 1
floor	number	Returns the number rounded down to the nearest integer
hypot	number1, number2	Returns sqrt(number1 ^ 2 + number2 ^ 2)
log	number	Returns the natural (base e) logarithm of number
log10	number	Returns the base-10 logarithm of number
log1p	number	Returns log10(1+number)
max	number1, number2	Returns the maximum of number1 or number2
min	number1, number2	Returns the minimum of number1 or number2
pow	number1, number2	Returns the result of number1 raised to the power of number2
sin	number	Returns the sine, with the parameter in radians
sinh	number	Returns the hyperbolic sine
sqrt	number	Returns the square root of number
tan	number	Returns the tangent, with the parameter in radians
tanh	number	Returns the hyperbolic tangent
degrees	number	Converts number from radians to degrees
radians	number	Converts number from degrees to radians

Suspension

There are two possible ways to suspend the execution of a Reflex script after it begins. Either the script can be suspended for a predetermined length of time, or a source script can suspend itself while it executes a target script, resuming when the target completes or fails. This practice is also known as script *coordination*.

Purpose of suspension

When a script in the Rapture environment is suspended, either directly or by running a target script, all variables, parameters, and other context information is frozen. The script can then be placed in the Rapture pipeline, or it can be assigned as a scheduled Rapture task to be executed at a fixed time.

One of the benefits of this feature is that the script can continue on a different Rapture server than the one on which it was started.

Functions for suspension

There are five functions for handling script suspension and execution. Note that the @ symbol at the start of a function name indicates that the function is asynchronous.

suspend

Syntax: `suspend(seconds)`
Description: This function simply suspends the script's execution for approximately the number of seconds passed as a parameter.

@call

Syntax: `@call(script, {parameters})`
Description: This function makes an asynchronous request to execute the `script`, along with any `parameters` that need to be passed. If the script does not take parameters, the curly brackets should be empty.

The request is placed onto the Rapture pipeline for execution, and the function returns a handle to the request. This handle can be used in conjunction with `@status` and `@wait`, if desired.
Example:

```
program = "println('I am the target script.')";
handle = @call(program, {});
```

@callscript

Syntax: `@callscript(partition, script, {parameters})`
Description: For scripts that are already hosted on a Rapture server, it is necessary to use `@callscript` instead of `@call` to run the target, so that the hosting Rapture partition can be identified. In all other respects, this function is identical to `@call`.

@status

Syntax: `@status(handle)`
Description: This function returns a block of metadata containing the real-time status of a target script that has been triggered by @call or @callscript.
Example:

```
program = "println('I am the target script.')";
handle = @call(program, {});
println(@status(handle));
```

The output of this example would be similar to the following:

```
{
state=COMPLETED,
taskId=2abd8d30-f7ea-4956-908c-e04737d64c0e,
relatedTaskId=,
creationTime=1354632237825,
startExecutionTime=1354632237827,
endExecutionTime=1354632237829,
suspensionCount=0,
output=["I am the target script."]
}
```

@wait

Syntax: `@wait(seconds, handles[])`
Description: This function waits for a sequential array of script handles to complete or fail. The `seconds` parameter suspends the execution of the source script between completions before it wakes up to check the status of the pending target scripts.

In the following example, all 50 scripts must complete before their statuses are printed.

Example:

```
handles = [] ;

for i = 1 to 50 do
  program = " println('hello from " + i + " ');";
  handle = @call (program, {}) ;
  handles = handles + handle;
end

@wait(10, handles) ;

for  h in handles do
 println(@status(h)) ;
end
```

Scripting Methods

Reflex scripts can be served either from a web server or directly from Rapture. Details of these methods follow.

Reflex from a web server

Serving Reflex scripts from a web server requires the `ReflexScriptPageServlet`. This servlet can be bound to a file suffix (typically `.rfx`) and is configured in the web.xml file. A typical configuration is shown here.

```
<servlet>
  <servlet-name>REFLEX</servlet-name>
<servlet-class>rapture.server.web.servlet.ReflexScriptPageServlet
</servlet-class>
  <init-param>
    <param-name>resourcePath</param-name>
    <param-value>/</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>REFLEX</servlet-name>
  <url-pattern>*.rfx</url-pattern>
</servlet-mapping>
```

The servlet loads the Reflex script from a URI and appends any script parameters in the standard URI format. Any output from `println` is sent to the HTTP client.

The code that follows is an example that runs from a web server. It prints out a JSON-formatted string that contains entries for all features installed in a Rapture environment.

```
// Returns the list of features
features = #feature.getInstalledFeatures () ;
ret = [];
  for feature in features do
  inner = {};
  inner[' feature'] = feature ['feature'];
  inner ['description'] = feature ['description'];
  ver = feature ['version'];
  inner ['version'] = ver ['major '] + '.' + ver ['minor'] + '.'
+ ver ['release'];
  ret = ret + inner;
end
println(json(ret));
```

Reflex from Rapture

Serving Reflex scripts from Rapture requires the `ReflexRefScriptPageServlet`. Rather than using a URI, this servlet is configured to reference a path to the script on a Rapture partition. After the script is loaded, its execution behaves in the same way as it does when served from the web.

The following servlet configuration code is typical for Reflex scripts served from Rapture.

```
<servlet>
  <servlet-name>REFLEXREF</servlet-name>
<servlet-class>rapture.server.web.servlet.ReflexRefScriptPageServ
let
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>REFLEXREF</servlet-name>
  <url-pattern>*.rrfx</url-pattern>
</servlet-mapping>
```

Appendix: Reflex Standard Library

This appendix lists the complete set of built-in functions that are usable in Reflex scripts.

NOTE: Reflex treats all the functions listed here as keywords. Therefore, it is not legal to create, for example, a variable that has the same name as one of these functions.

all

Syntax: `boolean all(booleanFunction, listExpression)`
Description: The `all` function takes a previously defined boolean function and tests it against every item in `listExpression`. It returns `true` if all items in `listExpression` are tested as true.
Example:

```
def isThree(val)
  return val == 3;
end

inputList1 = [1,2,3,4];
inputList2 = [3,3,3];
result1 = all(isThree,inputList1);
result2 = all(isThree,inputList2);
assert(result1 == false);
assert(result2 == true);
```

any

Syntax: `boolean any(booleanFunction, listExpression)`
Description: The `any` function takes a previously defined boolean function and tests it against every item in `listExpression`. It returns `true` if any item in `listExpression` is tested as true. (The function will stop testing as soon as it evaluates a true item.)
Example:

```
def isThree(val)
  return val == 3;
end

inputList1 = [1,2,3,4];
inputList2 = [2,4,6];
result1 = any(isThree,inputList1);
result2 = any(isThree,inputList2);
assert(result1 == true);
assert(result2 == false);
```

archive

Syntax: `file archive(parameters)`
Description: The `archive` function opens an existing WinZIP-compatible file, or creates one if the file is not found. Reflex map objects can be written to and read from an archive file with the standard Reflex push and pull operators.

The following examples illustrate how objects are first written to, and then read from, an archive.

Examples:

```
arcFile = archive("test.zip");

dataEntry1 = {"dataField1": 42, "data2": "A string"};
dataEntry2 = {"dataField1": 34, "data3": "A different string "};

dataEntry1 --> arcFile;
 ["DataEntryTwo", dataEntry2] --> arcFile;

close(arcFile);


arcFile = archive("test.zip");

dataRecord1 <--  arcFile;
dataRecord2 <--  arcFile;

close(arcFile);

println("First record data is " + dataRecord1['data']);
println("Second record data is " + dataRecord2['data']);
```

assert

Syntax: `assert(booleanExpression)`

Description: The `assert` function is used mostly for simple testing and debugging. If `booleanExpression` evaluates to `false`, the script terminates immediately with an error.

Example:

```
a = 2 + 3;
assert(a == 5); // and the script continues to run
```

call

Syntax: `object call(libExpression, stringExpression, mapExpression)`

Description: Once a third-party library is loaded with the `lib` function, the `call` function runs a specific method from within that library.

The `libExpression` is the name of the library, the `stringExpression` is the name of the method, and `mapExpression` contains the parameters passed to the method. The return value depends on the code in the method itself.

Example:

```
mylib = lib('rapture.test');

result = call(mylib, 'testFn', {'param' : 42});
```

capabilities

Syntax: `map capabilities()`
Description: This function returns a key-value pair for the current instance of Rapture, containing a string for each capability to indicate whether that capability is present.

The possible values for capabilities are:

- `CACHE`
- `DATA`
- `DEBUG`
- `IO`
- `OUTPUT`
- `PORT`
- `SCRIPT`
- `SUSPEND`

cast

Syntax: `object cast(targetExpression, "string" | "number")`
Description: The `cast` function attempts to coerce the value in `targetExpression` from a string to a number or vice versa. Internally, this function uses the `toString` method and its own number parser.
Example:

```
a = "1.0";
b = cast(a, "number");
assert(a == 1.0);

y = 1.0;
z = cast(y , "string");
assert(z == '1.0');
```

chain

Syntax: `object chain(scriptExpression [,mapExpression])`
Description: The `chain` function invokes another script and returns whatever return value applies to that script (or `void`). If the script takes parameters, the optional `mapExpression` can be used for passing them.
Example:

```
a = "println('The parameter is ' + p); return true;";

res = chain(a,{'p' : 42});

println("The result is " + res);
```

The output of this example would be:

```
The parameter is 42
The result is true
```

close

Syntax: `close(sourceExpression)`
Description: This function closes a previously opened file or port.

copy

Syntax: `copy(sourceStream, targetStream)`
Description: The copy function takes data from any stream-based source and transfers it to a stream-based target, overwriting any content previously in `targetStream`.

date

Syntax: `date date([stringExpression])`
Description: This function returns a date object. Without `stringExpression`, `date` returns the current date. The format used in `stringExpression` must be yyyyMMdd.
Example:

```
declareDate = date('17760704');
println("The Declaration of Independence was signed on " +
declareDate);
```

debug

Syntax: `debug(expression)`
Description: The debug function behaves in the same way as `println`, except that it directs `expression` to the debugger console instead of the usual output handler.

defined

Syntax: `boolean defined(identifier)`
Description: The `defined` function returns `true` if `identifier` has previously been used in the Reflex script.
Example:

```
a = 4;
assert(defined(a) == true);
assert(defined(b) == false);
```

delete

Syntax: `delete(stringExpression | fileExpression)`
Description: This function deletes either a file from the system or a Reflex file object. Be sure to test that the file exists before attempting to call `delete`.

difference

Syntax: `list difference(list1, list2)`
Description: The difference function returns a list of all unique elements from `list1` and `list2` that are not common to both lists. It works on lists of numbers or lists of strings.
Example:

```
a = [1,2,3];
b = [3,4,5];
c = difference(a, b); // c contains [1,2,4,5]
```

dropwhile

Syntax: `list dropwhile(booleanFunction, listExpression)`
Description: The `dropwhile` function uses a previously defined function to test every element in `listExpression`, and removes the item from the output list until it reaches an item for which the test evaluates to `false`.
Example:

```
def isNotThree(val)
  return val != 3;
end

inputList = [1,2,3,4];

result = dropwhile(isNotThree,inputList);
assert(result == [3,4]);
```

evals

Syntax: `string evals(stringExpression)`
Description: The evals function attempts to expand embedded variables in a quoted string, returning a regular string as output.
Example:

```
legs = 8;
evalstring = evals("Spiders have ${legs} legs.");
assert(evalstring == "Spiders have 8 legs.");
```

file

Syntax: `file file(stringExpression)`
Description: This function creates a Reflex `file` object, where `stringExpression` is a reference to an existing file or folder. The resulting file object can be written to or read from with the push and pull operators, as the example demonstrates.
Example:

```
a = "/tmp/test.txt";
data = "This is some text\n";

aFile = file(a);

data --> aFile;

b = "/tmp/test.txt";
bFile = file(b);
```

```
data2 <--  bFile;
assert(data == data2);
```

filter

Syntax: `list filter(filteringFunction,listExpression)`
Description: The `filter` function applies a user-defined, boolean `filteringFunction` to each item in `listExpression`, copying the item to the returned list if the filtering returns `true`.
Example:

```
def filtering(val)
  return val % 2 == 0;
end

inputList = [1,2,3,4];
result = filterFn(filtering,inputList);
assert(result == [2,4]);
```

fold

Syntax: `object fold(foldingFunction, initialExpression, listExpression)`
Description: This function applies a user-defined `foldingfunction` to each item in `listExpression`, storing the result in an accumulator. The folding function must contain exactly two parameters: the current accumulator and the current item in the list. The accumulator is set to `initialExpression` before the first folding function is run. When all items in the list have been folded into the accumulator value, `fold` returns the result.
Example:

```
def foldingsum(current, listVal)
  return current + listVal;
end

inputList = [1,2,3,4];
result = fold(foldingsum,0,inputList);
assert(result == 10);
```

format

Syntax: `string format(embeddedString, var [,var]...)`
Description: The `format` function attempts to expand string and number variables represented by %s and %d placeholders in `embeddedString`.
Example:

```
a = 25;
thismonth = December;
result = format("Today is %s %dth", thismonth, a);
assert(result == "Today is December 25th.");
```

fromjson

Syntax: `map fromjson(stringExpression)`
Description: This function creates a Reflex map object from a JSON-formatted string.
Example:

```
a = '{"one" : 1, "two" : 2}';
b = fromjson(a);
assert(b['one'] == 1);
```

getch

Syntax: `string getch()`
Description: The `getch` function retrieves the first typed character from standard input and places it into a single-character string. This function is currently supported only within ReflexRunner.

getln

Syntax: `string getln()`
Description The `getln` function retrieves typed characters from standard input until it receives a newline, and places all characters before the newline into a string. This function is currently supported only within ReflexRunner.

hascapability

Syntax: `boolean hascapability(capabilityString)`
Description: This function returns `true` if the current instance of Rapture contains the capability passed as the parameter.

The possible string values for capabilities are:

- `CACHE`
- `DATA`
- `DEBUG`
- `IO`
- `OUTPUT`
- `PORT`
- `SCRIPT`
- `SUSPEND`

import

Syntax: `import`
Description: The import function references a third-party module, so that its functions can be called from within the Reflex script. Refer to the Modules section for further details.

isfile

Syntax: `boolean isfile(stringExpression | fileExpression)`
Description: The `isfile` function returns `true` if a file represented by `expression` is found.

isfolder

Syntax: `boolean isfolder(stringExpression | fileExpression)`
Description: The `isfolder` function returns `true` if a folder represented by expression is found.

join

Syntax: `string join(string1, ..., stringN)`
`list join(object1, ..., objectN)`
Description: The join function accepts any type that can be used in a list, but returns a concatenated string if all the parameters are strings. Otherwise, it returns a single list of all the parameters.
Example:

```
assert(join(a,b,c) == 'abc');
assert(join(1,2,3) == [1,2,3]);
```

json

Syntax: `string json(mapExpression)`
Description: The json function converts a Reflex map object to a JSON-formatted string. (It does not deal directly with JSON documents.) It is the inverse of the `fromjson` function.
Example:

```
a = {'one' : 1, 'two' : 2};

a1 = " " + a;
a2 = json(a) ;

assert (a1 == '{one=1, two=2}');
assert (a2 == '{"one" : 1, "two" : 2}');
```

keys

Syntax: `list keys(mapExpression)`
Description: The keys function extracts the key from each key-value pair in a Reflex map, storing all the keys as a Reflex list of strings.
Example:

```
a = {'one':1, 'two':2};
b = keys(a); // b == ['one', 'two]
for k in b do
  println("Key " + k + " value is " + b[k]);
end
```

lib

Syntax: `lib lib(stringExpression)`
Description: The purpose of the lib function is to link a third-party library to Reflex. The `stringExpression` must reference a loadable class that implements the `IReflexLibrary` interface.
IMPORTANT: The third-party add-in needs to be on `/classpath`, along with any dependencies it may have.
Example:

```
mylib = lib('rapture.addins.BloombergData');
```

mapFn

Syntax: `list mapFn(mappingFunction,listExpression)`
Description: The `mapFn` function applies a user-defined mapping function on each item in `listExpression`, copying the mapped result into the output list.
Example:

```
def mapping(val)
  return val*7;
end

inputList = [1,2,3,4];
result = mapFn(mapping,inputList);
assert(result == [7,14,21,28]);
```

md5

Syntax: `string md5(string)`
Description: The md5 function returns an md5 hash of its string parameter. Strings in Rapture that contain sensitive information (such as passwords) must always be hashed before being passed over a non-secure link.

merge

Syntax: `map merge(mapExpression, mapExpression, [,...])`
Description: The merge function takes two or more maps and returns a single map of merged key-value pairs from the parameters. If two or more maps have the same key, the value in the merged map will be taken from the rightmost map in the parameter list. When the maps are nested, any lower-level maps will be merged recursively.
Examples:

```
a = { 'one' : 1 };
b = { 'two' : 2 };
c = merge(a, b);
assert(c == { 'one' : 1, 'two' : 2 });

d = { 'one' : 1 };
e = { 'one' : 'uno' };
f = merge(d, e);
assert(f == { 'one' : 'uno' });

g = { 'inner' : { 'one' : 1 }};
h = { 'inner' : { 'two' : 2 }};
```

```
i = merge(g,h);
assert(i == {'inner' : { 'one' : 1, 'two' : 2 }});
```

mergeif

Syntax: `map mergeif(mapExpression, mapExpression, [,...])`
Description: The `mergeif` function behaves identically to the `merge` function, except that it does not overwrite an existing value if more than one map has the same key-value pair.
Examples:

```
a = { 'one' : 1 };
b = { 'two' : 2 };
c = mergeif(a, b);
assert(c == { 'one' : 1, 'two' : 2 });

d = { 'one' : 1 };
e = { 'one' : 'uno' };
f = mergeif(d, e);
assert(f == { 'one' : '1' });
```

message

Syntax: `message(providerId,messageId)`
Description: This function takes a message with `messageId` from the message queue on the active Rapture server, and copies the message to another server with `providerId`.

mkdir

Syntax: `mkdir(pathExpression)`
Description: The mkdir function behaves similarly to the shell command. The pathExpression parameter is a string that can specify an absolute or relative path.

print

Syntax: `print(expression)`
Description: The print function behaves exactly as its `println` counterpart, described below, except that it does not append a newline character at the end of `expression`.
Example:

```
print("Hello, world!");
print(" These words are on the same line.");
println("");
```

println

Syntax: `println(expression)`
Description: The `println` function first takes each element of `expression`, coerces it into a string, and concatenates the string, appending a newline character. Finally, the

string is sent to the registered output handler, which is typically standard output, the Eclipse console, or a log file.

Example: The following statements are all legal uses of `println`:

```
println("Hello, world!");
println(5);
println({}); //prints an empty map
println("Rap" + "ture");
```

rand

Syntax: `number rand(numberExpression)`

Description: The `rand` function returns a randomly generated integer between zero and the value of `numberExpression`, inclusive.

readdir

Syntax: `list readdir(pathExpression)`

Description: The `readdir` function returns a list of files and subfolders in the directory that `pathExpression` denotes. The following example demonstrates a function that recursively prints out the names of all subfolders in a path.

Example:

```
def readFolder(folder)
  println("Looking at " + folder);
  filesAndFolders = readdir(folder);
  for fAndf in filesAndFolders do
    if isfolder(fAndf) do
      readFolder(fAndf);
    end
  end
end

readFolder('/tmp');
```

remove

Syntax: `remove(mapExpression,mapKeyExpression)`

Description: This function takes the map object given by `mapExpression`, and removes the map entry with the key that matches `mapKeyExpression`.

Example:

```
a = {};
a['one'] = 1;
a['two'] = 2;
assert(size(keys(a)) == 2);
remove(a,'one');
assert(size(keys(a) == 1);
assert(a['one'] == null);
```

replace

Syntax: `string replace(stringExpression,oldsubstrExpression, newsubstrExpression)`
`list replace(listExpression,oldLilstItemExpression, newListItemExpression)`
Description: The replace function can operate on either strings or lists. For strings, it searches `stringExpression` for every occurrence of `oldsubstrExpression` and replaces it with the string given by `newsubstrExpression`. For lists, the behavior is identical, replacing any old list items in `listExpression` with the new item in the parameter set.
Example:

```
a = "I am a Java developer.";
b = replace(a,"Java","Reflex");
assert(b == "I am a Reflex developer.");
```

round

Syntax: `number round(numberExpression)`
Description: If `numberExpression` is a floating point value, `round` returns the closest rounded integer, either up or down.

rpull

Syntax: `object rpull(sourceId)`
Description: The `rpull` function is an extension of the pull operator that is designed for documents, series data, and sheets only. It returns document data as a map, a sparse matrix containing all the elements of a series, or a sparse matrix containing all the cell values in a sheet, depending on the type of parameter data.

rpush

Syntax: `rpush(targetId,dataExpression)`
Description: The `rpush` function is the inverse of `rpull`, writing map data to documents or matrix data to a series or a sheet.

size

Syntax: `number size(listExpression | stringExpression)`
Description: The `size` function takes only one parameter, either a string or a list, and returns the parameter's length. Note that `size(null)` returns 0.
Example:

```
a = [0,3,7,6];
assert(size(a) == 4);
```

sleep

Syntax: `sleep(integerExpression)`
Description: The `sleep` function pauses the Reflex script for approximately the number of milliseconds given by `integerExpression`.

spawn

Syntax: `process spawn(listExpression [,mapExpression, fileExpression])`
Description: This function spawns a child process, where `listExpression` contains the command to launch the process, along with any flags or other parameters used by the process. The environment of the process can be specified with `mapExpression`, and `fileExpression` identifies the folder under which the process is run.

The process object that `spawn` returns can use the pull operation to retrieve its standard output, and the `wait` function can be used to determine when the process has ended.

The spawn function is available only in environments where spawning is supported, such as ReflexRunner. Spawning is not supported on the Rapture server.

Example:

```
env = {"PATH" : "/bin"};
folder = file ('/tmp') ;
program = ['/bin/ls', '-l'];

p = spawn(program, env, folder);

wait(p);

out <--  p;

println("Output from process is " + out);
```

split

Syntax: `list split(stringExpression,separator,parseBoolean)`
Description: This function splits the string given by stringExpression into multiple strings, copying each substring to the output list. The separator parameter is a character that indicates where to split the substring. If `parseBoolean` is true, and if the separating character appears in a quote, the string that quotes it will not be split.
Example:

```
a = '"Here, I","sit"';
b = split(a,',',false);
assert(b == ["Here","I","sit"]);
c = split(a,',',true);
assert(c == ["Here, I","sit"]);
```

splitwith

Syntax: `list splitwith(booleanFunction, listExpression)`
Description: This function splits `listExpression` into two lists and places both into a nested list as output. The splitting point is determined by a previously defined boolean function that tests each item in `listExpression`. The second list begins with the first item in `listExpression` for which this test returns `false`.
Example:

```
def isNotThree(val)
  return val != 3;
end

inputList = [1,2,3,4];

result = splitwith(isNotThree,inputList);
assert(result == [[1,2],[3,4]]);
```

takewhile

Syntax: `list takewhile(booleanFunction, listExpression)`
Description: The `takewhile` function uses a previously defined boolean function to test each item in `listExpression`, and it copies these items to the output list for as long as each item's test evaluates to `true`. Copying stops as soon as an item's test evaluates to `false`.
Example:

```
def isNotThree(val)
  return val != 3;
end

inputList = [1,2,3,4];

result = takewhile(isNotThree,inputList);
assert(result == [1,2]);
```

template

Syntax: `string template(stringExpression,mapExpression)`
Description: The `template` function takes a string as its input that behaves as a "template," with variables denoted by angle brackets (< >). The `mapExpression` parameter contains the names and values of each variable, and `template` returns a regular string with each of these variables filled in.
Example:

```
tmp = "Hello, <who>!";
vars = {'who' : 'world'};
val = template(tmp, vars);
assert(val == "Hello, world!");
```

time

Syntax: `time time([stringExpression])`
Description: This function returns a time object. Without `stringExpression`, `time` returns the current time on the system clock. The `stringExpression` must be a time formatted as HH:mm:ss. The time function is complementary to the `date` function, covered earlier.

timer

Syntax: `number timer([timerStart])`
Description: The `timer` function acts as a stopwatch that measures the elapsed time between its calls. Without parameters, the stopwatch begins counting and returns the start time. If called with a start time parameter, the timer returns the amount of time elapsed since it was last started or reset.
Example:

```
a = timer();
sleep(1000);
b = timer(a);
println("Elapsed time is " b);
```

transpose

Syntax: `matrix transpose(sourceMatrix)`
Description: The `transpose` function transposes the rows and columns of the `sourceMatrix` and copies the result into the output.
Example:

```
a[7,4] = 'string';
b = transpose(a);
assert(b[4,7] == 'string');
```

typeof

Syntax: `string typeof(expression)`
Description: Returns the name of `expression`'s data type, as defined in this document's section on Data Types. If `expression` has no type, `typeof` returns "void." If `expression` has a type that is not included in the list of internal types from Table 1, `typeof` returns "object."
Example:

```
a = "This is a string ";

if typeof(a) == "string" do
  println("Yes, 'a' is a string");
end
```

unique

Syntax: `list unique(list1, list2)`
Description: This function is identical to the `difference` function.

urldecode

Syntax: `string urldecode(sourceExpression)`
Description: This function takes a `sourceExpression` string that represents a percent-encoded URL and copies it to the output, replacing all percent-encoded characters with their normal ASCII values.

urlencode

Syntax: `string urlencode(sourceExpression)`
Description: This function takes a sourceExpression representing a normal URL and copies it to its output, replacing any unsafe ASCII characters with their percent-encoded equivalents.

use

Syntax: `use(remoteName)`
Description: The `use` function opens a Rapture "Remote," which connects one Rapture cloud environment to another. The server admin is responsible for creating and removing Remotes.

uuid

Syntax: `string uuid()`
Description: The `uuid` function generates a new, unique string.
Example:

```
a = uuid();
b = uuid();
assert(a != b);
```

vars

Syntax: `string vars(sourceVariable)`
Description: The `vars` function returns the context of the variable passed in through `sourceVariable`. Possible results are "local," "global," and "const."

wait

Syntax: `map wait(document [,interval ,count])`
`void wait(process)`
Description: The `wait` function has two uses. Most commonly, it checks whether `document` exists in Rapture, returning a map of the document's contents or `null` if the document does not exist. The `interval` parameter specifies how long to wait until checking for a document's existence again, and the `count` parameter specifies how many intervals to wait.
In addition, wait can be used to pause a script's processing until a spawned child process, indicated by a `process` object parameter, is complete. Refer to the `spawn` function for an example.