BuiltIn

Library version:	3.2.1
Library scope:	GLOBAL
Named arguments:	supported

Introduction

An always available standard library with often needed keywords.

`BuiltIn` is Robot Framework's standard library that provides a set of generic keywords needed often. It is imported automatically and thus always available. The provided keywords can be used, for example, for verifications (e.g. *Should Be Equal*, *Should Contain*), conversions (e.g. *Convert To Integer*) and for various other purposes (e.g. *Log*, *Sleep*, *Run Keyword If*, *Set Global Variable*).

Table of contents

HTML error messages

Many of the keywords accept an optional error message to use if the keyword fails, and it is possible to use HTML in these messages by prefixing them with `*HTML*`. See *Fail* keyword for a usage example. Notice that using HTML in messages is not limited to BuiltIn library but works with any error message.

Evaluating expressions

Many keywords, such as *Evaluate*, *Run Keyword If* and *Should Be True*, accept an expression that is evaluated in Python.

Evaluation namespace

Expressions are evaluated using Python's eval function so that all Python built-ins like `len()` and `int()` are available. In addition to that, all unrecognized variables are considered to be modules that are automatically imported. It is possible to use all available Python modules, including the standard modules and the installed third party modules.

Examples:

Should Be True	len('${result}') > 3	
Run Keyword If	os.sep == '/'	Non-Windows Keyword
${robot version} =	*Evaluate*	robot.__version__

Evaluate also allows configuring the execution namespace with a custom namespace and with custom modules to be imported. The latter functionality is useful when using nested modules like `rootmod.submod` that are implemented so that the root module does not automatically import sub modules. Otherwise the automatic module import mechanism described earlier is enough to get the needed modules imported.

NOTE: Automatic module import is a new feature in Robot Framework 3.2. Earlier modules needed to be explicitly taken into use when using the *Evaluate* keyword and other keywords only had access to `sys` and `os` modules.

Using variables

When a variable is used in the expressing using the normal `${variable}` syntax, its value is replaced before the expression is evaluated. This means that the value used in the expression will be the string representation of the variable value, not the variable value itself. This is not a problem with numbers and other objects that have a string representation that can be evaluated directly, but with other objects the behavior depends on the string representation. Most importantly, strings must always be quoted, and if they can contain newlines, they must be triple quoted.

Examples:

Should Be True	${rc} < 10	Return code greater than 10	
Run Keyword If	'${status}' == 'PASS'	Log	Passed
Run Keyword If	'FAIL' in '''${output}'''	Log	Output contains FAIL

Actual variables values are also available in the evaluation namespace. They can be accessed using special variable syntax without the curly braces like `$variable`. These variables should never be quoted.

Examples:

Should Be True	$rc < 10	Return code greater than 10	
Run Keyword If	$status == 'PASS'	*Log*	Passed
Run Keyword If	'FAIL' in $output	*Log*	Output contains FAIL
Should Be True	len($result) > 1 and $result[1] == 'OK'		
Should Be True	$result is not None		

Using the `$variable` syntax slows down expression evaluation a little. This should not typically matter, but should be taken into account if complex expressions are evaluated often and there are strict time constrains.

Notice that instead of creating complicated expressions, it is often better to move the logic into a test library. That eases maintenance and can also enhance execution speed.

Boolean arguments

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-insensitively. Keywords verifying something that allow dropping actual and expected values from the possible error message also consider string `no values` to be false. Other strings are considered true unless the keyword documentation explicitly states otherwise, and other argument types are tested using the same rules as in Python.

True examples:

Should Be Equal	${x}	${y}	Custom error	values=True	# Strings are generally true.
Should Be Equal	${x}	${y}	Custom error	values=yes	# Same as the above.
Should Be Equal	${x}	${y}	Custom error	values=${TRUE}	# Python `True` is true.
Should Be Equal	${x}	${y}	Custom error	values=${42}	# Numbers other than 0 are true.

False examples:

Should Be Equal	${x}	${y}	Custom error	values=False	# String `false` is false.
Should Be Equal	${x}	${y}	Custom error	values=no	# Also string `no` is false.
Should Be Equal	${x}	${y}	Custom error	values=${EMPTY}	# Empty string is false.
Should Be Equal	${x}	${y}	Custom error	values=${FALSE}	# Python `False` is false.
Should Be Equal	${x}	${y}	Custom error	values=no values	# `no values` works with `values` argument

Considering string `NONE` false is new in Robot Framework 3.0.3 and considering also `OFF` and `0` false is new in Robot Framework 3.1.

Pattern matching

Many keywords accepts arguments as either glob or regular expression patterns.

Glob patterns

Some keywords, for example *Should Match*, support so called glob patterns where:

`*`	matches any string, even an empty string
`?`	matches any single character
`[chars]`	matches one character in the bracket
`[!chars]`	matches one character not in the bracket
`[a-z]`	matches one character from the range in the bracket
`[!a-z]`	matches one character not from the range in the bracket

Unlike with glob patterns normally, path separator characters `/` and `\` and the newline character `\n` are matches by the above wildcards.

Support for brackets like `[abc]` and `[!a-z]` is new in Robot Framework 3.1.

Regular expressions

Some keywords, for example *Should Match Regexp*, support regular expressions that are more powerful but also more complicated that glob patterns. The regular expression support is implemented using Python's re module and its documentation should be consulted for more information about the syntax.

Because the backslash character (`\`) is an escape character in Robot Framework test data, possible backslash characters in regular expressions need to be escaped with another backslash like `\\d\\w+`. Strings that may contain special characters but should be handled as literal strings, can be escaped with the *Regexp Escape* keyword.

Multiline string comparison

Should Be Equal and *Should Be Equal As Strings* report the failures using unified diff format if both strings have more than two lines.

Example:

${first} =	*Catenate*	SEPARATOR=\n	Not in second	Same	Differs	Same
${second} =	*Catenate*	SEPARATOR=\n	Same	Differs2	Same	Not in first
Should Be Equal	${first}	${second}				

Results in the following error message:

```
Multiline strings are different:
--- first
+++ second
@@ -1,4 +1,4 @@
-Not in second
 Same
-Differs
+Differs2
 Same
+Not in first
```

String representations

Several keywords log values explicitly (e.g. *Log*) or implicitly (e.g. *Should Be Equal* when there are failures). By default keywords log values using "human readable" string representation, which means that strings like `Hello` and numbers like `42` are logged as-is. Most of the time this is the desired behavior, but there are some problems as well:

- It is not possible to see difference between different objects that have same string representation like string `42` and integer `42`. *Should Be Equal* and some other keywords add the type information to the error message in these cases, though.
- Non-printable characters such as the null byte are not visible.
- Trailing whitespace is not visible.
- Different newlines (`\r\n` on Windows, `\n` elsewhere) cannot be separated from each others.
- There are several Unicode characters that are different but look the same. One example is the Latin `a` (`\u0061`) and the Cyrillic `а` (`\u0430`). Error messages like `a != а` are not very helpful.
- Some Unicode characters can be represented using different forms. For example, `ä` can be represented either as a single code point `\u00e4` or using two code points `\u0061` and `\u0308` combined together. Such forms are considered canonically equivalent, but strings containing them are not considered equal when compared in Python. Error messages like `ä != ä` are not that helpful either.
- Containers such as lists and dictionaries are formatted into a single line making it hard to see individual items they contain.

To overcome the above problems, some keywords such as *Log* and *Should Be Equal* have an optional `formatter` argument that can be used to configure the string representation. The supported values are `str` (default), `repr`, and `ascii` that work similarly as Python built-in functions with same names. More detailed semantics are explained below.

The `formatter` argument is new in Robot Framework 3.1.2.

str

Use the "human readable" string representation. Equivalent to using `str()` in Python 3 and `unicode()` in Python 2. This is the default.

repr

Use the "machine readable" string representation. Similar to using `repr()` in Python, which means that strings like `Hello` are logged like `'Hello'`, newlines and non-printable characters are escaped like `\n` and `\x00`, and so on. Non-ASCII characters are shown as-is like `ä` in Python 3 and in escaped format like `\xe4` in Python 2. Use `ascii` to always get the escaped format.

There are also some enhancements compared to the standard `repr()`:

- Bigger lists, dictionaries and other containers are pretty-printed so that there is one item per row.
- On Python 2 the `u` prefix is omitted with Unicode strings and the `b` prefix is added to byte strings.

ascii

Same as using `ascii()` in Python 3 or `repr()` in Python 2 where `ascii()` does not exist. Similar to using `repr` explained above but with the following differences:

- On Python 3 non-ASCII characters are escaped like `\xe4` instead of showing them as-is like `ä`. This makes it easier to see differences between Unicode characters that look the same but are not equal. This is how `repr()` works in Python 2.
- On Python 2 just uses the standard `repr()` meaning that Unicode strings get the `u` prefix and no `b` prefix is added to byte strings.
- Containers are not pretty-printed.

Shortcuts

Keywords

<table>
<tr><th>Keyword</th><th>Arguments</th><th>Documentation</th></tr>
<tr><td>Call Method</td><td>object, method_name, *args, **kwargs</td><td>Calls the named method of the given object with the provided arguments.

The possible return value from the method is returned and can be assigned to a variable. Keyword fails both if the object does not have a method with the given name or if executing the method raises an exception.

Possible equal signs in arguments must be escaped with a backslash like <code>\=</code>.

Examples:
<table><tr><td>Call Method</td><td>${hashtable}</td><td>put</td><td>myname</td><td>myvalue</td></tr><tr><td>${isempty} =</td><td>Call Method</td><td>${hashtable}</td><td>isEmpty</td><td></td></tr><tr><td>Should Not Be True</td><td>${isempty}</td><td></td><td></td><td></td></tr><tr><td>${value} =</td><td>Call Method</td><td>${hashtable}</td><td>get</td><td>myname</td></tr><tr><td>Should Be Equal</td><td>${value}</td><td>myvalue</td><td></td><td></td></tr><tr><td>Call Method</td><td>${object}</td><td>kwargs</td><td>name=value</td><td>foo=bar</td></tr><tr><td>Call Method</td><td>${object}</td><td>positional</td><td>escaped\=equals</td><td></td></tr></table></td></tr>
<tr><td>Catenate</td><td>*items</td><td>Catenates the given items together and returns the resulted string.

By default, items are catenated with spaces, but if the first item contains the string <code>SEPARATOR=<sep></code>, the separator <code><sep></code> is used instead. Items are converted into strings when necessary.

Examples:
<table><tr><td>${str1} =</td><td>Catenate</td><td>Hello</td><td>world</td><td></td></tr><tr><td>${str2} =</td><td>Catenate</td><td>SEPARATOR=---</td><td>Hello</td><td>world</td></tr><tr><td>${str3} =</td><td>Catenate</td><td>SEPARATOR=</td><td>Hello</td><td>world</td></tr></table>=>
<pre>${str1} = 'Hello world'
${str2} = 'Hello---world'
${str3} = 'Helloworld'</pre></td></tr>
<tr><td>Comment</td><td>*messages</td><td>Displays the given messages in the log file as keyword arguments.

This keyword does nothing with the arguments it receives, but as they are visible in the log, this keyword can be used to display simple messages. Given arguments are ignored so thoroughly that they can even contain non-existing variables. If you are interested about variable values, you can use the Log or Log Many keywords.</td></tr>
<tr><td>Continue For Loop</td><td></td><td>Skips the current for loop iteration and continues from the next.

Skips the remaining keywords in the current for loop iteration and continues from the next one. Can be used directly in a for loop or in a keyword that the loop uses.

Example:
<table><tr><td>:FOR</td><td>${var}</td><td>IN</td><td>@{VALUES}</td></tr><tr><td></td><td>Run Keyword If</td><td>'${var}' == 'CONTINUE'</td><td>Continue For Loop</td></tr><tr><td></td><td>Do Something</td><td>${var}</td><td></td></tr></table>See Continue For Loop If to conditionally continue a for loop without using Run Keyword If or other wrapper keywords.</td></tr>
<tr><td>Continue For Loop If</td><td>condition</td><td>Skips the current for loop iteration if the <code>condition</code> is true.

A wrapper for Continue For Loop to continue a for loop based on the given condition. The condition is evaluated using the same semantics as with Should Be True keyword.

Example:
<table><tr><td>:FOR</td><td>${var}</td><td>IN</td><td>@{VALUES}</td></tr><tr><td></td><td>Continue For Loop If</td><td>'${var}' == 'CONTINUE'</td><td></td></tr><tr><td></td><td>Do Something</td><td>${var}</td><td></td></tr></table></td></tr>
<tr><td>Convert To Binary</td><td>item, base=None, prefix=None, length=None</td><td>Converts the given item to a binary string.

The <code>item</code>, with an optional <code>base</code>, is first converted to an integer using Convert To Integer internally. After that it is converted to a binary number (base 2) represented as a string such as <code>1011</code>.

The returned value can contain an optional <code>prefix</code> and can be required to be of minimum <code>length</code> (excluding the prefix and a possible minus sign). If the value is initially shorter than the required length, it is padded with zeros.

Examples:
<table><tr><td>${result} =</td><td>Convert To Binary</td><td>10</td><td></td><td></td><td># Result is 1010</td></tr><tr><td>${result} =</td><td>Convert To Binary</td><td>F</td><td>base=16</td><td>prefix=0b</td><td># Result is 0b1111</td></tr><tr><td>${result} =</td><td>Convert To Binary</td><td>-2</td><td>prefix=B</td><td>length=4</td><td># Result is -B0010</td></tr></table>See also Convert To Integer, Convert To Octal and Convert To Hex.</td></tr>
<tr><td>Convert To Boolean</td><td>item</td><td>Converts the given item to Boolean true or false.

Handles strings <code>True</code> and <code>False</code> (case-insensitive) as expected, otherwise returns item's truth value using Python's <code>bool()</code> method.</td></tr>
<tr><td>Convert To Bytes</td><td>input, input_type=text</td><td>Converts the given <code>input</code> to bytes according to the <code>input_type</code>.

Valid input types are listed below:

▪ <code>text:</code> Converts text to bytes character by character. All characters with ordinal below 256 can be used and are converted to bytes with same values. Many characters are easiest to represent using escapes like <code>\x00</code> or <code>\xff</code>. Supports both Unicode strings and bytes.

▪ <code>int:</code> Converts integers separated by spaces to bytes. Similarly as with Convert To Integer, it is possible to use binary, octal, or hex values by prefixing the values with <code>0b</code>, <code>0o</code>, or <code>0x</code>, respectively.</td></tr>
</table>

- `hex:` Converts hexadecimal values to bytes. Single byte is always two characters long (e.g. `01` or `FF`). Spaces are ignored and can be used freely as a visual separator.
- `bin:` Converts binary values to bytes. Single byte is always eight characters long (e.g. `00001010`). Spaces are ignored and can be used freely as a visual separator.

In addition to giving the input as a string, it is possible to use lists or other iterables containing individual characters or numbers. In that case numbers do not need to be padded to certain length and they cannot contain extra spaces.

Examples (last column shows returned bytes):

${bytes} =	Convert To Bytes	hyvä		# hyv\xe4
${bytes} =	Convert To Bytes	\xff\x07		# \xff\x07
${bytes} =	Convert To Bytes	82 70	int	# RF
${bytes} =	Convert To Bytes	0b10 0x10	int	# \x02\x10
${bytes} =	Convert To Bytes	ff 00 07	hex	# \xff\x00\x07
${bytes} =	Convert To Bytes	5246212121	hex	# RF!!!
${bytes} =	Convert To Bytes	0000 1000	bin	# \x08
${input} =	Create List	1	2	12
${bytes} =	Convert To Bytes	${input}	int	# \x01\x02\x0c
${bytes} =	Convert To Bytes	${input}	hex	# \x01\x02\x12

Use *Encode String To Bytes* in `String` library if you need to convert text to bytes using a certain encoding.

Convert To Hex — *item*, *base=None*, *prefix=None*, *length=None*, *lowercase=False*

Converts the given item to a hexadecimal string.

The `item`, with an optional `base`, is first converted to an integer using *Convert To Integer* internally. After that it is converted to a hexadecimal number (base 16) represented as a string such as `FF0A`.

The returned value can contain an optional `prefix` and can be required to be of minimum `length` (excluding the prefix and a possible minus sign). If the value is initially shorter than the required length, it is padded with zeros.

By default the value is returned as an upper case string, but the `lowercase` argument a true value (see *Boolean arguments*) turns the value (but not the given prefix) to lower case.

Examples:

${result} =	Convert To Hex	255			# Result is FF
${result} =	Convert To Hex	-10	prefix=0x	length=2	# Result is -0x0A
${result} =	Convert To Hex	255	prefix=X	lowercase=yes	# Result is Xff

See also *Convert To Integer*, *Convert To Binary* and *Convert To Octal*.

Convert To Integer — *item*, *base=None*

Converts the given item to an integer number.

If the given item is a string, it is by default expected to be an integer in base 10. There are two ways to convert from other bases:

- Give base explicitly to the keyword as `base` argument.
- Prefix the given string with the base so that `0b` means binary (base 2), `0o` means octal (base 8), and `0x` means hex (base 16). The prefix is considered only when `base` argument is not given and may itself be prefixed with a plus or minus sign.

The syntax is case-insensitive and possible spaces are ignored.

Examples:

${result} =	Convert To Integer	100		# Result is 100
${result} =	Convert To Integer	FF AA	16	# Result is 65450
${result} =	Convert To Integer	100	8	# Result is 64
${result} =	Convert To Integer	-100	2	# Result is -4
${result} =	Convert To Integer	0b100		# Result is 4
${result} =	Convert To Integer	-0x100		# Result is -256

See also *Convert To Number*, *Convert To Binary*, *Convert To Octal*, *Convert To Hex*, and *Convert To Bytes*.

Convert To Number — *item*, *precision=None*

Converts the given item to a floating point number.

If the optional `precision` is positive or zero, the returned number is rounded to that number of decimal digits. Negative precision means that the number is rounded to the closest multiple of 10 to the power of the absolute precision. If a number is equally close to a certain precision, it is always rounded away from zero.

Examples:

${result} =	Convert To Number	42.512		# Result is 42.512
${result} =	Convert To Number	42.512	1	# Result is 42.5
${result} =	Convert To Number	42.512	0	# Result is 43.0
${result} =	Convert To Number	42.512	-1	# Result is 40.0

Notice that machines generally cannot store floating point numbers accurately. This may cause surprises with these numbers in general and also when they are rounded. For more information see, for example, these resources:

- http://docs.python.org/tutorial/floatingpoint.html
- http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition

If you want to avoid possible problems with floating point numbers, you can implement custom keywords using Python's decimal or fractions modules.

If you need an integer number, use *Convert To Integer* instead.

Convert To Octal — *item*, *base=None*, *prefix=None*, *length=None*

Converts the given item to an octal string.

The `item`, with an optional `base`, is first converted to an integer using *Convert To Integer* internally. After that it is converted to an octal number (base 8) represented as a string such as `775`.

The returned value can contain an optional `prefix` and can be required to be of minimum `length` (excluding the prefix and a possible minus sign). If the value is initially shorter than the required length, it is padded with zeros.

Examples:

${result} =	Convert To Octal	10			# Result is 12
${result} =	Convert To Octal	-F	base=16	prefix=0	# Result is -017
${result} =	Convert To Octal	16	prefix=oct	length=4	# Result is oct0020

See also *Convert To Integer*, *Convert To Binary* and *Convert To Hex*.

Convert To String — *item*

Converts the given item to a Unicode string.

Strings are also NFC normalized.

Use *Encode String To Bytes* and *Decode Bytes To String* keywords in `String` library if you need to convert between Unicode and byte strings using different encodings. Use *Convert To Bytes* if you just want to create byte strings.

Create Dictionary — **items*

Creates and returns a dictionary based on the given `items`.

Items are typically given using the `key=value` syntax same way as `&{dictionary}` variables are created in the Variable table. Both keys and values can contain variables, and possible equal sign in key can be escaped with a backslash like `escaped\=key=value`. It is also possible to get items from existing dictionaries by simply using them like `&{dict}`.

Alternatively items can be specified so that keys and values are given separately. This and the `key=value` syntax can even be combined, but separately given items must be first. If same key is used multiple times, the last value has precedence.

The returned dictionary is ordered, and values with strings as keys can also be accessed using a convenient dot-access syntax like `${dict.key}`. Technically the returned dictionary is Robot Framework's own `DotDict` instance. If there is a need, it can be converted into a regular Python `dict` instance by using the *Convert To Dictionary* keyword from the Collections library.

Examples:

&{dict} =	Create Dictionary	key=value	foo=bar			# key=value syntax
Should Be True	${dict} == {'key': 'value', 'foo': 'bar'}					
&{dict2} =	Create Dictionary	key	value	foo	bar	# separate key and value
Should Be Equal	${dict}	${dict2}				
&{dict} =	Create Dictionary	${1}=${2}	&{dict}	foo=new		# using variables
Should Be True	${dict} == {1: 2, 'key': 'value', 'foo': 'new'}					
Should Be Equal	${dict.key}	value				# dot-access

Create List **items*

Returns a list containing given items.

The returned list can be assigned both to `${scalar}` and `@{list}` variables.

Examples:

@{list} =	Create List	a	b	c
${scalar} =	Create List	a	b	c
${ints} =	Create List	${1}	${2}	${3}

Evaluate *expression*, *modules=None*, *namespace=None*

Evaluates the given expression in Python and returns the result.

`expression` is evaluated in Python as explained in the *Evaluating expressions* section.

`modules` argument can be used to specify a comma separated list of Python modules to be imported and added to the evaluation namespace.

`namespace` argument can be used to pass a custom evaluation namespace as a dictionary. Possible `modules` are added to this namespace.

Starting from Robot Framework 3.2, modules used in the expression are imported automatically. `modules` argument is still needed with nested modules like `rootmod.submod` that are implemented so that the root module does not automatically import sub modules. This is illustrated by the `selenium.webdriver` example below.

Variables used like `${variable}` are replaced in the expression before evaluation. Variables are also available in the evaluation namespace and can be accessed using the special `$variable` syntax as explained in the *Evaluating expressions* section.

Examples (expecting `${result}` is number 3.14):

${status} =	Evaluate	0 < ${result} < 10	# Would also work with string '3.14'
${status} =	Evaluate	0 < $result < 10	# Using variable itself, not string representation
${random} =	Evaluate	random.randint(0, sys.maxsize)	
${options} =	Evaluate	selenium.webdriver.ChromeOptions()	modules=selenium.webdriver
${ns} =	Create Dictionary	x=${4}	y=${2}
${result} =	Evaluate	x*10 + y	namespace=${ns}

=>

```
${status} = True
${random} = <random integer>
${options} = ChromeOptions instance
${result} = 42
```

NOTE: Prior to Robot Framework 3.2 using `modules=rootmod.submod` was not enough to make the root module itself available in the evaluation namespace. It needed to be taken into use explicitly like `modules=rootmod, rootmod.submod`.

Exit For Loop

Stops executing the enclosing for loop.

Exits the enclosing for loop and continues execution after it. Can be used directly in a for loop or in a keyword that the loop uses.

Example:

:FOR	${var}	IN	@{VALUES}
	Run Keyword If	'${var}' == 'EXIT'	Exit For Loop
	Do Something	${var}	

See *Exit For Loop If* to conditionally exit a for loop without using *Run Keyword If* or other wrapper keywords.

Exit For Loop If *condition*

Stops executing the enclosing for loop if the `condition` is true.

A wrapper for *Exit For Loop* to exit a for loop based on the given condition. The condition is evaluated using the same semantics as with *Should Be True* keyword.

Example:

:FOR	${var}	IN	@{VALUES}
	Exit For Loop If	'${var}' == 'EXIT'	
	Do Something	${var}	

Fail *msg=None*, **tags*

Fails the test with the given message and optionally alters its tags.

The error message is specified using the `msg` argument. It is possible to use HTML in the given error message, similarly as with any other keyword accepting an error message, by prefixing the error with `*HTML*`.

It is possible to modify tags of the current test case by passing tags after the message. Tags starting with a hyphen (e.g. `-regression`) are removed and others added. Tags are modified using *Set Tags* and *Remove Tags* internally, and the semantics setting and removing them are the same as with these keywords.

Examples:

Fail	Test not ready			# Fails with the given message.
Fail	*HTML*Test not ready			# Fails using HTML in the message.
Fail	Test not ready	not-ready		# Fails and adds 'not-ready' tag.
Fail	OS not supported	-regression		# Removes tag 'regression'.
Fail	My message	tag	-t*	# Removes all tags starting with 't' except the newly added 'tag'.

See *Fatal Error* if you need to stop the whole test execution.

Fatal Error *msg=None*

Stops the whole test execution.

		The test or suite where this keyword is used fails with the provided message, and subsequent tests fail with a canned message. Possible teardowns will nevertheless be executed. See *Fail* if you only want to stop one test case unconditionally.
Get Count	*container*, *item*	Returns and logs how many times `item` is found from `container`. This keyword works with Python strings and lists and all objects that either have `count` method or can be converted to Python lists. Example: | ${count} = | Get Count | ${some item} | interesting value | | Should Be True | 5 < ${count} < 10 | | |
Get Length	*item*	Returns and logs the length of the given item as an integer. The item can be anything that has a length, for example, a string, a list, or a mapping. The keyword first tries to get the length with the Python function `len`, which calls the item's `__len__` method internally. If that fails, the keyword tries to call the item's possible `length` and `size` methods directly. The final attempt is trying to get the value of the item's `length` attribute. If all these attempts are unsuccessful, the keyword fails. Examples: | ${length} = | Get Length | Hello, world! | | | Should Be Equal As Integers | ${length} | 13 | | | @{list} = | Create List | Hello, | world! | | ${length} = | Get Length | ${list} | | | Should Be Equal As Integers | ${length} | 2 | | See also *Length Should Be*, *Should Be Empty* and *Should Not Be Empty*.
Get Library Instance	*name=None*, *all=False*	Returns the currently active instance of the specified test library. This keyword makes it easy for test libraries to interact with other test libraries that have state. This is illustrated by the Python example below: `from robot.libraries.BuiltIn import BuiltIn` `def title_should_start_with(expected):` `  seleniumlib = BuiltIn().get_library_instance('SeleniumLibrary')` `  title = seleniumlib.get_title()` `  if not title.startswith(expected):` `  raise AssertionError("Title '%s' did not start with '%s'"` `  % (title, expected))` It is also possible to use this keyword in the test data and pass the returned library instance to another keyword. If a library is imported with a custom name, the `name` used to get the instance must be that name and not the original library name. If the optional argument `all` is given a true value, then a dictionary mapping all library names to instances will be returned. Example: | &{all libs} = | Get library instance | all=True |
Get Time	*format=timestamp*, *time_=NOW*	Returns the given time in the requested format. **NOTE:** DateTime library contains much more flexible keywords for getting the current date and time and for date and time handling in general. How time is returned is determined based on the given `format` string as follows. Note that all checks are case-insensitive. 1) If `format` contains the word `epoch`, the time is returned in seconds after the UNIX epoch (1970-01-01 00:00:00 UTC). The return value is always an integer. 2) If `format` contains any of the words `year`, `month`, `day`, `hour`, `min`, or `sec`, only the selected parts are returned. The order of the returned parts is always the one in the previous sentence and the order of words in `format` is not significant. The parts are returned as zero-padded strings (e.g. May -> `05`). 3) Otherwise (and by default) the time is returned as a timestamp string in the format `2006-02-24 15:08:31`. By default this keyword returns the current local time, but that can be altered using `time` argument as explained below. Note that all checks involving strings are case-insensitive. 1) If `time` is a number, or a string that can be converted to a number, it is interpreted as seconds since the UNIX epoch. This documentation was originally written about 1177654467 seconds after the epoch. 2) If `time` is a timestamp, that time will be used. Valid timestamp formats are `YYYY-MM-DD hh:mm:ss` and `YYYYMMDD hhmmss`. 3) If `time` is equal to `NOW` (default), the current local time is used. 4) If `time` is equal to `UTC`, the current time in UTC is used. 5) If `time` is in the format like `NOW - 1 day` or `UTC + 1 hour 30 min`, the current local/UTC time plus/minus the time specified with the time string is used. The time string format is described in an appendix of Robot Framework User Guide. Examples (expecting the current local time is 2006-03-29 15:06:21): | ${time} = | Get Time | | | | | ${secs} = | Get Time | epoch | | | | ${year} = | Get Time | return year | | | | ${yyyy} | ${mm} | ${dd} = | Get Time | year,month,day | | @{time} = | Get Time | year month day hour min sec | | | | ${y} | ${s} = \| Get Time \| seconds and year \| \| => `${time} = '2006-03-29 15:06:21'` `${secs} = 1143637581` `${year} = '2006'` `${yyyy} = '2006', ${mm} = '03', ${dd} = '29'` `@{time} = ['2006', '03', '29', '15', '06', '21']` `${y} = '2006'` `${s} = '21'` Examples (expecting the current local time is 2006-03-29 15:06:21 and UTC time is 2006-03-29 12:06:21): | ${time} = | Get Time | | 1177654467 | # Time given as epoch seconds | | ${secs} = | Get Time | sec | 2007-04-27 09:14:27 | # Time given as a timestamp | | ${year} = | Get Time | year | NOW | # The local time of execution | | @{time} = | Get Time | hour min sec | NOW + 1h 2min 3s | # 1h 2min 3s added to the local time |

		@{utc} = Get Time hour min sec UTC # The UTC time of execution ${hour} = Get Time hour UTC - 1 hour # 1h subtracted from the UTC time => `${time} = '2007-04-27 09:14:27'` `${secs} = 27` `${year} = '2006'` `@{time} = ['16', '08', '24']` `@{utc} = ['12', '06', '21']` `${hour} = '11'`
Get Variable Value	*name*, *default=None*	Returns variable value or `default` if the variable does not exist. The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*. Examples: ${x} = Get Variable Value ${a} default ${y} = Get Variable Value ${a} ${b} ${z} = Get Variable Value ${z} => `${x} gets value of ${a} if ${a} exists and string 'default' otherwise` `${y} gets value of ${a} if ${a} exists and value of ${b} otherwise` `${z} is set to Python None if it does not exist previously` See *Set Variable If* for another keyword to set variables dynamically.
Get Variables	*no_decoration=False*	Returns a dictionary containing all variables in the current scope. Variables are returned as a special dictionary that allows accessing variables in space, case, and underscore insensitive manner similarly as accessing variables in the test data. This dictionary supports all same operations as normal Python dictionaries and, for example, Collections library can be used to access or modify it. Modifying the returned dictionary has no effect on the variables available in the current scope. By default variables are returned with `${}`, `@{}` or `&{}` decoration based on variable types. Giving a true value (see *Boolean arguments*) to the optional argument `no_decoration` will return the variables without the decoration. Example: ${example_variable} = Set Variable example value ${variables} = Get Variables Dictionary Should Contain Key ${variables} \${example_variable} Dictionary Should Contain Key ${variables} \${ExampleVariable} Set To Dictionary ${variables} \${name} value Variable Should Not Exist \${name} ${no decoration} = Get Variables no_decoration=Yes Dictionary Should Contain Key ${no decoration} example_variable
Import Library	*name*, **args*	Imports a library with the given name and optional arguments. This functionality allows dynamic importing of libraries while tests are running. That may be necessary, if the library itself is dynamic and not yet available when test data is processed. In a normal case, libraries should be imported using the Library setting in the Setting table. This keyword supports importing libraries both using library names and physical paths. When paths are used, they must be given in absolute format or found from search path. Forward slashes can be used as path separators in all operating systems. It is possible to pass arguments to the imported library and also named argument syntax works if the library supports it. `WITH NAME` syntax can be used to give a custom name to the imported library. Examples: Import Library MyLibrary Import Library ${CURDIR}/../Library.py arg1 named=arg2 Import Library ${LIBRARIES}/Lib.java arg WITH NAME JavaLib
Import Resource	*path*	Imports a resource file with the given path. Resources imported with this keyword are set into the test suite scope similarly when importing them in the Setting table using the Resource setting. The given path must be absolute or found from search path. Forward slashes can be used as path separator regardless the operating system. Examples: Import Resource ${CURDIR}/resource.txt Import Resource ${CURDIR}/../resources/resource.html Import Resource found_from_pythonpath.robot
Import Variables	*path*, **args*	Imports a variable file with the given path and optional arguments. Variables imported with this keyword are set into the test suite scope similarly when importing them in the Setting table using the Variables setting. These variables override possible existing variables with the same names. This functionality can thus be used to import new variables, for example, for each test in a test suite. The given path must be absolute or found from search path. Forward slashes can be used as path separator regardless the operating system. Examples: Import Variables ${CURDIR}/variables.py Import Variables ${CURDIR}/../vars/env.py arg1 arg2 Import Variables file_from_pythonpath.py
Keyword Should Exist	*name*, *msg=None*	Fails unless the given keyword exists in the current scope. Fails also if there are more than one keywords with the same name. Works both with the short name (e.g. `Log`) and the full name (e.g. `BuiltIn.Log`). The default error message can be overridden with the `msg` argument. See also *Variable Should Exist*.
Length Should Be	*item*, *length*, *msg=None*	Verifies that the length of the given item is correct. The length of the item is got using the *Get Length* keyword. The default error message can be overridden with the `msg` argument.
Log	*message*,	Logs the given message with the given level.

	level=INFO, *html=False*, *console=False*, *repr=False*, *formatter=str*	Valid levels are TRACE, DEBUG, INFO (default), HTML, WARN, and ERROR. Messages below the current active log level are ignored. See *Set Log Level* keyword and `--loglevel` command line option for more details about setting the level. Messages logged with the WARN or ERROR levels will be automatically visible also in the console and in the Test Execution Errors section in the log file. If the `html` argument is given a true value (see *Boolean arguments*), the message will be considered HTML and special characters such as `<` are not escaped. For example, logging `<img src="image.png">` creates an image when `html` is true, but otherwise the message is that exact string. An alternative to using the `html` argument is using the HTML pseudo log level. It logs the message as HTML using the INFO level. If the `console` argument is true, the message will be written to the console where test execution was started from in addition to the log file. This keyword always uses the standard output stream and adds a newline after the written message. Use *Log To Console* instead if either of these is undesirable, The `formatter` argument controls how to format the string representation of the message. Possible values are `str` (default), `repr` and `ascii`, and they work similarly to Python built-in functions with same names. When using `repr`, bigger lists, dictionaries and other containers are also pretty-printed so that there is one item per row. For more details see *String representations*. This is a new feature in Robot Framework 3.1.2. The old way to control string representation was using the `repr` argument, and `repr=True` is still equivalent to using `formatter=repr`. The `repr` argument will be deprecated in the future, though, and using `formatter` is thus recommended. Examples: | Log | Hello, world! | | | # Normal INFO message. | | Log | Warning, world! | WARN | | # Warning. | | Log | Hello, world! | html=yes | | # INFO message as HTML. | | Log | Hello, world! | HTML | | # Same as above. | | Log | Hello, world! | DEBUG | html=true | # DEBUG as HTML. | | Log | Hello, console! | console=yes | | # Log also to the console. | | Log | Null is \x00 | formatter=repr | | # Log `'Null is \x00'`. | See *Log Many* if you want to log multiple messages in one go, and *Log To Console* if you only want to write to the console.
Log Many	**messages*	Logs the given messages as separate entries using the INFO level. Supports also logging list and dictionary variable items individually. Examples: | Log Many | Hello | ${var} | | Log Many | @{list} | &{dict} | See *Log* and *Log To Console* keywords if you want to use alternative log levels, use HTML, or log to the console.
Log To Console	*message*, *stream=STDOUT*, *no_newline=False*	Logs the given message to the console. By default uses the standard output stream. Using the standard error stream is possibly by giving the `stream` argument value `STDERR` (case-insensitive). By default appends a newline to the logged message. This can be disabled by giving the `no_newline` argument a true value (see *Boolean arguments*). Examples: | Log To Console | Hello, console! | | | Log To Console | Hello, stderr! | STDERR | | Log To Console | Message starts here and is | no_newline=true | | Log To Console | continued without newline. | | This keyword does not log the message to the normal log file. Use *Log* keyword, possibly with argument `console`, if that is desired.
Log Variables	*level=INFO*	Logs all variables in the current scope with given log level.
No Operation		Does absolutely nothing.
Pass Execution	*message*, **tags*	Skips rest of the current test, setup, or teardown with PASS status. This keyword can be used anywhere in the test data, but the place where used affects the behavior: ▪ When used in any setup or teardown (suite, test or keyword), passes that setup or teardown. Possible keyword teardowns of the started keywords are executed. Does not affect execution or statuses otherwise. ▪ When used in a test outside setup or teardown, passes that particular test case. Possible test and keyword teardowns are executed. Possible continuable failures before this keyword is used, as well as failures in executed teardowns, will fail the execution. It is mandatory to give a message explaining why execution was passed. By default the message is considered plain text, but starting it with `*HTML*` allows using HTML formatting. It is also possible to modify test tags passing tags after the message similarly as with *Fail* keyword. Tags starting with a hyphen (e.g. `-regression`) are removed and others added. Tags are modified using *Set Tags* and *Remove Tags* internally, and the semantics setting and removing them are the same as with these keywords. Examples: | Pass Execution | All features available in this version tested. | | | | Pass Execution | Deprecated test. | deprecated | -regression | This keyword is typically wrapped to some other keyword, such as *Run Keyword If*, to pass based on a condition. The most common case can be handled also with *Pass Execution If*: | Run Keyword If | ${rc} < 0 | Pass Execution | Negative values are cool. | | Pass Execution If | ${rc} < 0 | Negative values are cool. | | Passing execution in the middle of a test, setup or teardown should be used with care. In the worst case it leads to tests that skip all the parts that could actually uncover problems in the tested application. In cases where execution cannot continue do to external factors, it is often safer to fail the test case and make it non-critical.
Pass Execution If	*condition*, *message*, **tags*	Conditionally skips rest of the current test, setup, or teardown with PASS status. A wrapper for *Pass Execution* to skip rest of the current test, setup or teardown based the given `condition`. The condition is evaluated similarly as with *Should Be True* keyword, and `message` and `*tags` have same semantics as with *Pass Execution*. Example: | :FOR | ${var} \| IN \| @{VALUES} \| \| \| Pass Execution If \| '${var}' == 'EXPECTED' | Correct value was found | | | Do Something | ${var} | |
Regexp Escape	**patterns*	Returns each argument string escaped for use as a regular expression. This keyword can be used to escape strings to be used with *Should Match Regexp* and *Should Not Match Regexp* keywords.

Escaping is done with Python's `re.escape()` function.

Examples:

${escaped} =	Regexp Escape	${original}
@{strings} =	Regexp Escape	@{strings}

Reload Library *name_or_instance*

Rechecks what keywords the specified library provides.

Can be called explicitly in the test data or by a library itself when keywords it provides have changed.

The library can be specified by its name or as the active instance of the library. The latter is especially useful if the library itself calls this keyword as a method.

Remove Tags **tags*

Removes given `tags` from the current test or all tests in a suite.

Tags can be given exactly or using a pattern with `*`, `?` and `[chars]` acting as wildcards. See the *Glob patterns* section for more information.

This keyword can affect either one test case or all test cases in a test suite similarly as *Set Tags* keyword.

The current tags are available as a built-in variable `@{TEST TAGS}`.

Example:

Remove Tags	mytag	something-*	?ython

See *Set Tags* if you want to add certain tags and *Fail* if you want to fail the test case after setting and/or removing tags.

Repeat Keyword *repeat*, *name*, **args*

Executes the specified keyword multiple times.

`name` and `args` define the keyword that is executed similarly as with *Run Keyword*. `repeat` specifies how many times (as a count) or how long time (as a timeout) the keyword should be executed.

If `repeat` is given as count, it specifies how many times the keyword should be executed. `repeat` can be given as an integer or as a string that can be converted to an integer. If it is a string, it can have postfix `times` or `x` (case and space insensitive) to make the expression more explicit.

If `repeat` is given as timeout, it must be in Robot Framework's time format (e.g. `1 minute`, `2 min 3 s`). Using a number alone (e.g. `1` or `1.5`) does not work in this context.

If `repeat` is zero or negative, the keyword is not executed at all. This keyword fails immediately if any of the execution rounds fails.

Examples:

Repeat Keyword	5 times	Go to Previous Page		
Repeat Keyword	${var}	Some Keyword	arg1	arg2
Repeat Keyword	2 minutes	Some Keyword	arg1	arg2

Specifying `repeat` as a timeout is new in Robot Framework 3.0.

Replace Variables *text*

Replaces variables in the given text with their current values.

If the text contains undefined variables, this keyword fails. If the given `text` contains only a single variable, its value is returned as-is and it can be any object. Otherwise this keyword always returns a string.

Example:

The file `template.txt` contains `Hello ${NAME}!` and variable `${NAME}` has the value `Robot`.

${template} =	Get File	${CURDIR}/template.txt
${message} =	Replace Variables	${template}
Should Be Equal	${message}	Hello Robot!

Return From Keyword **return_values*

Returns from the enclosing user keyword.

This keyword can be used to return from a user keyword with PASS status without executing it fully. It is also possible to return values similarly as with the `[Return]` setting. For more detailed information about working with the return values, see the User Guide.

This keyword is typically wrapped to some other keyword, such as *Run Keyword If* or *Run Keyword If Test Passed*, to return based on a condition:

Run Keyword If	${rc} < 0	Return From Keyword
Run Keyword If Test Passed	Return From Keyword	

It is possible to use this keyword to return from a keyword also inside a for loop. That, as well as returning values, is demonstrated by the *Find Index* keyword in the following somewhat advanced example. Notice that it is often a good idea to move this kind of complicated logic into a test library.

```
*** Variables ***
@{LIST} =    foo    baz

*** Test Cases ***
Example
    ${index} =    Find Index    baz    @{LIST}
    Should Be Equal    ${index}    ${1}
    ${index} =    Find Index    non existing    @{LIST}
    Should Be Equal    ${index}    ${-1}

*** Keywords ***
Find Index
   [Arguments]    ${element}    @{items}
   ${index} =    Set Variable    ${0}
   :FOR    ${item}    IN    @{items}
   \    Run Keyword If    '${item}' == '${element}'    Return From Keyword    ${index}
   \    ${index} =    Set Variable    ${index + 1}
   Return From Keyword    ${-1}    # Also [Return] would work here.
```

The most common use case, returning based on an expression, can be accomplished directly with *Return From Keyword If*. See also *Run Keyword And Return* and *Run Keyword And Return If*.

Return From Keyword If *condition*, **return_values*

Returns from the enclosing user keyword if `condition` is true.

A wrapper for *Return From Keyword* to return based on the given condition. The condition is evaluated using the same semantics as with *Should Be True* keyword.

Given the same example as in *Return From Keyword*, we can rewrite the *Find Index* keyword as follows:

```
*** Keywords ***
Find Index
   [Arguments]    ${element}    @{items}
```

```
${index} =    Set Variable    ${0}
:FOR    ${item}    IN    @{items}
\    Return From Keyword If    '${item}' == '${element}'    ${index}
\    ${index} =    Set Variable    ${index + 1}
Return From Keyword    ${-1}    # Also [Return] would work here.
```

See also *Run Keyword And Return* and *Run Keyword And Return If*.

Run Keyword	*name*, **args*	Executes the given keyword with the given arguments. Because the name of the keyword to execute is given as an argument, it can be a variable and thus set dynamically, e.g. from a return value of another keyword or from the command line.
Run Keyword And Continue On Failure	*name*, **args*	Runs the keyword and continues execution even if a failure occurs. The keyword name and arguments work as with *Run Keyword*. Example: Run Keyword And Continue On Failure | Fail | This is a stupid example Log | This keyword is executed The execution is not continued if the failure is caused by invalid syntax, timeout, or fatal exception.
Run Keyword And Expect Error	*expected_error*, *name*, **args*	Runs the keyword and checks that the expected error occurred. The keyword to execute and its arguments are specified using `name` and `*args` exactly like with *Run Keyword*. The expected error must be given in the same format as in Robot Framework reports. By default it is interpreted as a glob pattern with `*`, `?` and `[chars]` as wildcards, but starting from Robot Framework 3.1 that can be changed by using various prefixes explained in the table below. Prefixes are case-sensitive and they must be separated from the actual message with a colon and an optional space like `PREFIX: Message` or `PREFIX:Message`. **Prefix** | **Explanation** `EQUALS` | Exact match. Especially useful if the error contains glob wildcards. `STARTS` | Error must start with the specified error. `REGEXP` | Regular expression match. `GLOB` | Same as the default behavior. See the *Pattern matching* section for more information about glob patterns and regular expressions. If the expected error occurs, the error message is returned and it can be further processed or tested if needed. If there is no error, or the error does not match the expected error, this keyword fails. Examples: Run Keyword And Expect Error | My error | Keyword | arg Run Keyword And Expect Error | ValueError: * | Some Keyword Run Keyword And Expect Error | STARTS: ValueError: | Some Keyword Run Keyword And Expect Error | EQUALS:No match for '//input[@type="text"]' ... | Find Element | //input[@type="text"] ${msg} = | Run Keyword And Expect Error | * ... | Keyword | arg1 | arg2 Log To Console | ${msg} Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword.
Run Keyword And Ignore Error	*name*, **args*	Runs the given keyword with the given arguments and ignores possible error. This keyword returns two values, so that the first is either string `PASS` or `FAIL`, depending on the status of the executed keyword. The second value is either the return value of the keyword or the received error message. See *Run Keyword And Return Status* If you are only interested in the execution status. The keyword name and arguments work as in *Run Keyword*. See *Run Keyword If* for a usage example. Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword. Otherwise this keyword itself never fails.
Run Keyword And Return	*name*, **args*	Runs the specified keyword and returns from the enclosing user keyword. The keyword to execute is defined with `name` and `*args` exactly like with *Run Keyword*. After running the keyword, returns from the enclosing user keyword and passes possible return value from the executed keyword further. Returning from a keyword has exactly same semantics as with *Return From Keyword*. Example: *Run Keyword And Return* | *My Keyword* | arg1 | arg2 # Above is equivalent to: ${result} = | *My Keyword* | arg1 | arg2 *Return From Keyword* | ${result} Use *Run Keyword And Return If* if you want to run keyword and return based on a condition.
Run Keyword And Return If	*condition*, *name*, **args*	Runs the specified keyword and returns from the enclosing user keyword. A wrapper for *Run Keyword And Return* to run and return based on the given `condition`. The condition is evaluated using the same semantics as with *Should Be True* keyword. Example: *Run Keyword And Return If* | ${rc} > 0 | *My Keyword* | arg1 | arg2 # Above is equivalent to: *Run Keyword If* | ${rc} > 0 | *Run Keyword And Return* | *My Keyword* | arg1 | arg2 Use *Return From Keyword If* if you want to return a certain value based on a condition.
Run Keyword And Return Status	*name*, **args*	Runs the given keyword with given arguments and returns the status as a Boolean value. This keyword returns Boolean `True` if the keyword that is executed succeeds and `False` if it fails. This is useful, for example, in combination with *Run Keyword If*. If you are interested in the error message or return value, use *Run Keyword And Ignore Error* instead. The keyword name and arguments work as in *Run Keyword*. Example: ${passed} = | *Run Keyword And Return Status* | Keyword | args *Run Keyword If* | ${passed} | Another keyword Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword. Otherwise this keyword itself never fails.
Run Keyword If	*condition*, *name*, **args*	Runs the given keyword with the given arguments, if `condition` is true.

The given `condition` is evaluated in Python as explained in *Evaluating expressions*, and `name` and `*args` have same semantics as with *Run Keyword*.

Example, a simple if/else construct:

${status}	${value} =	*Run Keyword And Ignore Error*	*My Keyword*
Run Keyword If	'${status}' == 'PASS'	*Some Action*	arg
Run Keyword Unless	'${status}' == 'PASS'	*Another Action*	

In this example, only either *Some Action* or *Another Action* is executed, based on the status of *My Keyword*. Instead of *Run Keyword And Ignore Error* you can also use *Run Keyword And Return Status*.

Variables used like `${variable}`, as in the examples above, are replaced in the expression before evaluation. Variables are also available in the evaluation namespace and can be accessed using special syntax `$variable` as explained in the *Evaluating expressions* section.

Example:

Run Keyword If	$result is None or $result == 'FAIL'	*Keyword*

This keyword supports also optional ELSE and ELSE IF branches. Both of them are defined in `*args` and must use exactly format `ELSE` or `ELSE IF`, respectively. ELSE branches must contain first the name of the keyword to execute and then its possible arguments. ELSE IF branches must first contain a condition, like the first argument to this keyword, and then the keyword to execute and its possible arguments. It is possible to have ELSE branch after ELSE IF and to have multiple ELSE IF branches. Nested *Run Keyword If* usage is not supported when using ELSE and/or ELSE IF branches.

Given previous example, if/else construct can also be created like this:

${status}	${value} =	*Run Keyword And Ignore Error*	*My Keyword*		
Run Keyword If	'${status}' == 'PASS'	*Some Action*	arg	ELSE	*Another Action*

The return value of this keyword is the return value of the actually executed keyword or Python `None` if no keyword was executed (i.e. if `condition` was false). Hence, it is recommended to use ELSE and/or ELSE IF branches to conditionally assign return values from keyword to variables (see *Set Variable If* if you need to set fixed values conditionally). This is illustrated by the example below:

${var1} =	*Run Keyword If*	${rc} == 0	*Some keyword returning a value*		
...	ELSE IF	0 < ${rc} < 42	*Another keyword*		
...	ELSE IF	${rc} < 0	*Another keyword with args*	${rc}	arg2
...	ELSE	*Final keyword to handle abnormal cases*	${rc}		
${var2} =	*Run Keyword If*	${condition}	*Some keyword*		

In this example, ${var2} will be set to `None` if ${condition} is false.

Notice that `ELSE` and `ELSE IF` control words must be used explicitly and thus cannot come from variables. If you need to use literal `ELSE` and `ELSE IF` strings as arguments, you can escape them with a backslash like `\ELSE` and `\ELSE IF`.

Python's os and sys modules are automatically imported when evaluating the `condition`. Attributes they contain can thus be used in the condition:

Run Keyword If	os.sep == '/'	*Unix Keyword*	
...	ELSE IF	sys.platform.startswith('java')	*Jython Keyword*
...	ELSE	*Windows Keyword*	

Run Keyword If All Critical Tests Passed	*name*, **args*	Runs the given keyword with the given arguments, if all critical tests passed. This keyword can only be used in suite teardown. Trying to use it in any other place will result in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If All Tests Passed	*name*, **args*	Runs the given keyword with the given arguments, if all tests passed. This keyword can only be used in a suite teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If Any Critical Tests Failed	*name*, **args*	Runs the given keyword with the given arguments, if any critical tests failed. This keyword can only be used in a suite teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If Any Tests Failed	*name*, **args*	Runs the given keyword with the given arguments, if one or more tests failed. This keyword can only be used in a suite teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If Test Failed	*name*, **args*	Runs the given keyword with the given arguments, if the test failed. This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If Test Passed	*name*, **args*	Runs the given keyword with the given arguments, if the test passed. This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword If Timeout Occurred	*name*, **args*	Runs the given keyword if either a test or a keyword timeout has occurred. This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error. Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.
Run Keyword Unless	*condition*, *name*, **args*	Runs the given keyword with the given arguments if `condition` is false. See *Run Keyword If* for more information and an example. Notice that this keyword does not support `ELSE` or `ELSE IF` branches like *Run Keyword If* does, though.
Run Keywords	**keywords*	Executes all the given keywords in a sequence. This keyword is mainly useful in setups and teardowns when they need to take care of multiple actions and creating a new higher level user keyword would be an overkill. By default all arguments are expected to be keywords to be executed. Examples: *Run Keywords* | *Initialize database* | *Start servers* | *Clear logs* *Run Keywords* | ${KW 1} | ${KW 2} *Run Keywords* | @{KEYWORDS} Keywords can also be run with arguments using upper case `AND` as a separator between keywords. The keywords are executed so that the first argument is the first keyword and proceeding arguments until the first `AND` are arguments to it. First argument after the first `AND` is the second keyword and proceeding arguments until the next `AND` are its arguments. And so on.

		Examples: | *Run Keywords* | *Initialize database* | db1 | AND | *Start servers* | server1 | server2 | | | *Run Keywords* | *Initialize database* | ${DB NAME} | AND | *Start servers* | @{SERVERS} | AND | *Clear logs* | | *Run Keywords* | ${KW} | AND | @{KW WITH ARGS} | | | | Notice that the `AND` control argument must be used explicitly and cannot itself come from a variable. If you need to use literal `AND` string as argument, you can either use variables or escape it with a backslash like `\AND`.
Set Global Variable	*name*, **values*	Makes a variable available globally in all tests and suites. Variables set with this keyword are globally available in all subsequent test suites, test cases and user keywords. Also variables in variable tables are overridden. Variables assigned locally based on keyword return values or by using *Set Test Variable* and *Set Suite Variable* override these variables in that scope, but the global value is not changed in those cases. In practice setting variables with this keyword has the same effect as using command line options `--variable` and `--variablefile`. Because this keyword can change variables everywhere, it should be used with care. See *Set Suite Variable* for more information and examples.
Set Library Search Order	**search_order*	Sets the resolution order to use when a name matches multiple keywords. The library search order is used to resolve conflicts when a keyword name in the test data matches multiple keywords. The first library (or resource, see below) containing the keyword is selected and that keyword implementation used. If the keyword is not found from any library (or resource), test executing fails the same way as when the search order is not set. When this keyword is used, there is no need to use the long `LibraryName.Keyword Name` notation. For example, instead of having | MyLibrary.Keyword | arg | | MyLibrary.Another Keyword | | | MyLibrary.Keyword | xxx | you can have | Set Library Search Order | MyLibrary | | Keyword | arg | | Another Keyword | | | Keyword | xxx | This keyword can be used also to set the order of keywords in different resource files. In this case resource names must be given without paths or extensions like: | Set Library Search Order | resource | another_resource | **NOTE:** ▪ The search order is valid only in the suite where this keywords is used. ▪ Keywords in resources always have higher priority than keywords in libraries regardless the search order. ▪ The old order is returned and can be used to reset the search order later. ▪ Library and resource names in the search order are both case and space insensitive.
Set Local Variable	*name*, **values*	Makes a variable available everywhere within the local scope. Variables set with this keyword are available within the local scope of the currently executed test case or in the local scope of the keyword in which they are defined. For example, if you set a variable in a user keyword, it is available only in that keyword. Other test cases or keywords will not see variables set with this keyword. This keyword is equivalent to a normal variable assignment based on a keyword return value. Example: | @{list} = | Create List | item1 | item2 | item3 | is equivalent with | Set Local Variable | @{list} | item1 | item2 | item3 | This keyword will provide the option of setting local variables inside keywords like *Run Keyword If*, *Run Keyword And Return If*, *Run Keyword Unless* which until now was not possible by using *Set Variable*. It will also be possible to use this keyword from external libraries that want to set local variables. New in Robot Framework 3.2.
Set Log Level	*level*	Sets the log threshold to the specified level and returns the old level. Messages below the level will not logged. The default logging level is INFO, but it can be overridden with the command line option `--loglevel`. The available levels: TRACE, DEBUG, INFO (default), WARN, ERROR and NONE (no logging).
Set Suite Documentation	*doc*, *append=False*, *top=False*	Sets documentation for the current test suite. By default the possible existing documentation is overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword. This keyword sets the documentation of the current suite by default. If the optional `top` argument is given a true value (see *Boolean arguments*), the documentation of the top level suite is altered instead. The documentation of the current suite is available as a built-in variable `${SUITE DOCUMENTATION}`.
Set Suite Metadata	*name*, *value*, *append=False*, *top=False*	Sets metadata for the current test suite. By default possible existing metadata values are overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword. This keyword sets the metadata of the current suite by default. If the optional `top` argument is given a true value (see *Boolean arguments*), the metadata of the top level suite is altered instead. The metadata of the current suite is available as a built-in variable `${SUITE METADATA}` in a Python dictionary. Notice that modifying this variable directly has no effect on the actual metadata the suite has.
Set Suite Variable	*name*, **values*	Makes a variable available everywhere within the scope of the current suite. Variables set with this keyword are available everywhere within the scope of the currently executed test suite. Setting variables with this keyword thus has the same effect as creating them using the Variable table in the test data file or importing them from variable files. Possible child test suites do not see variables set with this keyword by default, but that can be controlled by using `children=<option>` as the last argument. If the specified `<option>` given a true value (see *Boolean arguments*), the variable is set also to the child suites. Parent and sibling suites will never see variables set with this keyword. The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format as `\${NAME}` or `$NAME`. Variable value can be given using the same syntax as when variables are created in the Variable table.

		If a variable already exists within the new scope, its value will be overwritten. Otherwise a new variable is created. If a variable already exists within the current scope, the value can be left empty and the variable within the new scope gets the value within the current scope. Examples: | Set Suite Variable | ${SCALAR} | Hello, world! | | | Set Suite Variable | ${SCALAR} | Hello, world! | children=true | | Set Suite Variable | @{LIST} | First item | Second item | | Set Suite Variable | &{DICT} | key=value | foo=bar | | ${ID} = | Get ID | | | | Set Suite Variable | ${ID} \| \| \| To override an existing value with an empty value, use built-in variables `${EMPTY}`, `@{EMPTY}` or `&{EMPTY}`: | Set Suite Variable | ${SCALAR} | ${EMPTY} | | Set Suite Variable | @{LIST} | @{EMPTY} | | Set Suite Variable | &{DICT} | &{EMPTY} | **NOTE:** If the variable has value which itself is a variable (escaped or not), you must always use the escaped format to set the variable: Example: | ${NAME} = | Set Variable | \${var} | | | Set Suite Variable | ${NAME} | value | # Sets variable ${var} | | Set Suite Variable | \${NAME} | value | # Sets variable ${NAME} | This limitation applies also to *Set Test Variable*, *Set Global Variable*, *Variable Should Exist*, *Variable Should Not Exist* and *Get Variable Value* keywords.
Set Tags	**tags*	Adds given `tags` for the current test or all tests in a suite. When this keyword is used inside a test case, that test gets the specified tags and other tests are not affected. If this keyword is used in a suite setup, all test cases in that suite, recursively, gets the given tags. It is a failure to use this keyword in a suite teardown. The current tags are available as a built-in variable `@{TEST TAGS}`. See *Remove Tags* if you want to remove certain tags and *Fail* if you want to fail the test case after setting and/or removing tags.
Set Task Variable	*name*, **values*	Makes a variable available everywhere within the scope of the current task. This is an alias for *Set Test Variable* that is more applicable when creating tasks, not tests. New in RF 3.1.
Set Test Documentation	*doc*, *append=False*	Sets documentation for the current test case. By default the possible existing documentation is overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword. The current test documentation is available as a built-in variable `${TEST DOCUMENTATION}`. This keyword can not be used in suite setup or suite teardown.
Set Test Message	*message*, *append=False*	Sets message for the current test case. If the optional `append` argument is given a true value (see *Boolean arguments*), the given `message` is added after the possible earlier message by joining the messages with a space. In test teardown this keyword can alter the possible failure message, but otherwise failures override messages set by this keyword. Notice that in teardown the message is available as a built-in variable `${TEST MESSAGE}`. It is possible to use HTML format in the message by starting the message with `*HTML*`. Examples: | Set Test Message | My message | | | Set Test Message | is continued. | append=yes | | Should Be Equal | ${TEST MESSAGE} | My message is continued. | | Set Test Message | *HTML* Hello! | | This keyword can not be used in suite setup or suite teardown.
Set Test Variable	*name*, **values*	Makes a variable available everywhere within the scope of the current test. Variables set with this keyword are available everywhere within the scope of the currently executed test case. For example, if you set a variable in a user keyword, it is available both in the test case level and also in all other user keywords used in the current test. Other test cases will not see variables set with this keyword. See *Set Suite Variable* for more information and examples.
Set Variable	**values*	Returns the given values which can then be assigned to a variables. This keyword is mainly used for setting scalar variables. Additionally it can be used for converting a scalar variable containing a list to a list variable or to multiple scalar variables. It is recommended to use *Create List* when creating new lists. Examples: | ${hi} = | Set Variable | Hello, world! | | | | ${hi2} = | Set Variable | I said: ${hi} | | | | ${var1} | ${var2} = | Set Variable | Hello | world | | @{list} = | Set Variable | ${list with some items} | | | | ${item1} | ${item2} = | Set Variable | ${list with 2 items} | | Variables created with this keyword are available only in the scope where they are created. See *Set Global Variable*, *Set Test Variable* and *Set Suite Variable* for information on how to set variables so that they are available also in a larger scope.
Set Variable If	*condition*, **values*	Sets variable based on the given condition. The basic usage is giving a condition and two values. The given condition is first evaluated the same way as with the *Should Be True* keyword. If the condition is true, then the first value is returned, and otherwise the second value is returned. The second value can also be omitted, in which case it has a default value None. This usage is illustrated in the examples below, where `${rc}` is assumed to be zero. | ${var1} = | Set Variable If | ${rc} == 0 | zero | nonzero | | ${var2} = | Set Variable If | ${rc} > 0 | value1 | value2 | | ${var3} = | Set Variable If | ${rc} > 0 \| whatever \| \| => `${var1} = 'zero'` `${var2} = 'value2'` `${var3} = None`

It is also possible to have 'else if' support by replacing the second value with another condition, and having two new values after it. If the first condition is not true, the second is evaluated and one of the values after it is returned based on its truth value. This can be continued by adding more conditions without a limit.

${var} =	Set Variable If	${rc} == 0	zero
...	${rc} > 0	greater than zero	less then zero
${var} =	Set Variable If		
...	${rc} == 0	zero	
...	${rc} == 1	one	
...	${rc} == 2	two	
...	${rc} > 2	greater than two	
...	${rc} < 0	less than zero	

Use *Get Variable Value* if you need to set variables dynamically based on whether a variable exist or not.

Should Be Empty — *item*, *msg=None*

Verifies that the given item is empty.

The length of the item is got using the *Get Length* keyword. The default error message can be overridden with the `msg` argument.

Should Be Equal — *first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *formatter=str*

Fails if the given objects are unequal.

Optional `msg`, `values` and `formatter` arguments specify how to construct the error message if this keyword fails:

- If `msg` is not given, the error message is `<first> != <second>`.
- If `msg` is given and `values` gets a true value (default), the error message is `<msg>: <first> != <second>`.
- If `msg` is given and `values` gets a false value (see *Boolean arguments*), the error message is simply `<msg>`.
- `formatter` controls how to format the values. Possible values are `str` (default), `repr` and `ascii`, and they work similarly as Python built-in functions with same names. See *String representations* for more details.

If `ignore_case` is given a true value (see *Boolean arguments*) and both arguments are strings, comparison is done case-insensitively. If both arguments are multiline strings, this keyword uses *multiline string comparison*.

Examples:

Should Be Equal	${x}	expected		
Should Be Equal	${x}	expected	Custom error message	
Should Be Equal	${x}	expected	Custom message	values=False
Should Be Equal	${x}	expected	ignore_case=True	formatter=repr

`ignore_case` and `formatter` are new features in Robot Framework 3.0.1 and 3.1.2, respectively.

Should Be Equal As Integers — *first*, *second*, *msg=None*, *values=True*, *base=None*

Fails if objects are unequal after converting them to integers.

See *Convert To Integer* for information how to convert integers from other bases than 10 using `base` argument or `0b/0o/0x` prefixes.

See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`.

Examples:

Should Be Equal As Integers	42	${42}	Error message
Should Be Equal As Integers	ABCD	abcd	base=16
Should Be Equal As Integers	0b1011	11	

Should Be Equal As Numbers — *first*, *second*, *msg=None*, *values=True*, *precision=6*

Fails if objects are unequal after converting them to real numbers.

The conversion is done with *Convert To Number* keyword using the given `precision`.

Examples:

Should Be Equal As Numbers	${x}	1.1		# Passes if ${x} is 1.1
Should Be Equal As Numbers	1.123	1.1	precision=1	# Passes
Should Be Equal As Numbers	1.123	1.4	precision=0	# Passes
Should Be Equal As Numbers	112.3	75	precision=-2	# Passes

As discussed in the documentation of *Convert To Number*, machines generally cannot store floating point numbers accurately. Because of this limitation, comparing floats for equality is problematic and a correct approach to use depends on the context. This keyword uses a very naive approach of rounding the numbers before comparing them, which is both prone to rounding errors and does not work very well if numbers are really big or small. For more information about comparing floats, and ideas on how to implement your own context specific comparison algorithm, see http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/.

If you want to avoid possible problems with floating point numbers, you can implement custom keywords using Python's decimal or fractions modules.

See *Should Not Be Equal As Numbers* for a negative version of this keyword and *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`.

Should Be Equal As Strings — *first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *formatter=str*

Fails if objects are unequal after converting them to strings.

See *Should Be Equal* for an explanation on how to override the default error message with `msg`, `values` and `formatter`.

If `ignore_case` is given a true value (see *Boolean arguments*), comparison is done case-insensitively. If both arguments are multiline strings, this keyword uses *multiline string comparison*.

Strings are always NFC normalized.

`ignore_case` and `formatter` are new features in Robot Framework 3.0.1 and 3.1.2, respectively.

Should Be True — *condition*, *msg=None*

Fails if the given condition is not true.

If `condition` is a string (e.g. `${rc} < 10`), it is evaluated as a Python expression as explained in *Evaluating expressions* and the keyword status is decided based on the result. If a non-string item is given, the status is got directly from its truth value.

The default error message (`<condition> should be true`) is not very informative, but it can be overridden with the `msg` argument.

Examples:

Should Be True	${rc} < 10	
Should Be True	'${status}' == 'PASS'	# Strings must be quoted
Should Be True	${number}	# Passes if ${number} is not zero
Should Be True	${list}	# Passes if ${list} is not empty

Variables used like `${variable}`, as in the examples above, are replaced in the expression before evaluation. Variables are also available in the evaluation namespace, and can be accessed using special `$variable` syntax as explained in the *Evaluating expressions* section.

Examples:

Should Be True	$rc < 10	
Should Be True	$status == 'PASS'	# Expected string must be quoted

Should Be True automatically imports Python's os and sys modules that contain several useful attributes:

Should Be True	os.linesep == '\n'	# Unixy
Should Be True	os.linesep == '\r\n'	# Windows
Should Be True	sys.platform == 'darwin'	# OS X
Should Be True	sys.platform.startswith('java')	# Jython

Should Contain

container, *item*, *msg=None*, *values=True*, *ignore_case=False*

Fails if `container` does not contain `item` one or more times.

Works with strings, lists, and anything that supports Python's `in` operator.

See *Should Be Equal* for an explanation on how to override the default error message with arguments `msg` and `values`.

If `ignore_case` is given a true value (see *Boolean arguments*) and compared items are strings, it indicates that comparison should be case-insensitive. If the `container` is a list-like object, string items in it are compared case-insensitively. New option in Robot Framework 3.0.1.

Examples:

Should Contain	${output}	PASS		
Should Contain	${some list}	value	msg=Failure!	values=False
Should Contain	${some list}	value	ignore_case=True	

Should Contain Any

container, **items*, ***configuration*

Fails if `container` does not contain any of the `*items`.

Works with strings, lists, and anything that supports Python's `in` operator.

Supports additional configuration parameters `msg`, `values` and `ignore_case`, which have exactly the same semantics as arguments with same names have with *Should Contain*. These arguments must always be given using `name=value` syntax after all `items`.

Note that possible equal signs in `items` must be escaped with a backslash (e.g. `foo\=bar`) to avoid them to be passed in as `**configuration`.

Examples:

Should Contain Any	${string}	substring 1	substring 2		
Should Contain Any	${list}	item 1	item 2	item 3	
Should Contain Any	${list}	item 1	item 2	item 3	ignore_case=True
Should Contain Any	${list}	@{items}	msg=Custom message	values=False	

New in Robot Framework 3.0.1.

Should Contain X Times

container, *item*, *count*, *msg=None*, *ignore_case=False*

Fails if `container` does not contain `item` `count` times.

Works with strings, lists and all objects that *Get Count* works with. The default error message can be overridden with `msg` and the actual count is always logged.

If `ignore_case` is given a true value (see *Boolean arguments*) and compared items are strings, it indicates that comparison should be case-insensitive. If the `container` is a list-like object, string items in it are compared case-insensitively. New option in Robot Framework 3.0.1.

Examples:

Should Contain X Times	${output}	hello	2	
Should Contain X Times	${some list}	value	3	ignore_case=True

Should End With

str1, *str2*, *msg=None*, *values=True*, *ignore_case=False*

Fails if the string `str1` does not end with the string `str2`.

See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.

Should Match

string, *pattern*, *msg=None*, *values=True*, *ignore_case=False*

Fails if the given `string` does not match the given `pattern`.

Pattern matching is similar as matching files in a shell with `*`, `?` and `[chars]` acting as wildcards. See the *Glob patterns* section for more information.

See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.

Should Match Regexp

string, *pattern*, *msg=None*, *values=True*

Fails if `string` does not match `pattern` as a regular expression.

See the *Regular expressions* section for more information about regular expressions and how to use then in Robot Framework test data.

Notice that the given pattern does not need to match the whole string. For example, the pattern `ello` matches the string `Hello world!`. If a full match is needed, the `^` and `$` characters can be used to denote the beginning and end of the string, respectively. For example, `^ello$` only matches the exact string `ello`.

Possible flags altering how the expression is parsed (e.g. `re.IGNORECASE`, `re.MULTILINE`) must be embedded to the pattern like `(?im)pattern`. The most useful flags are `i` (case-insensitive), `m` (multiline mode), `s` (dotall mode) and `x` (verbose).

If this keyword passes, it returns the portion of the string that matched the pattern. Additionally, the possible captured groups are returned.

See the *Should Be Equal* keyword for an explanation on how to override the default error message with the `msg` and `values` arguments.

Examples:

Should Match Regexp	${output}	\\d{6}	# Output contains six numbers
Should Match Regexp	${output}	^\\d{6}$	# Six numbers and nothing more
${ret} =	Should Match Regexp	Foo: 42	(?i)foo: \\d+
${match}	${group1}	${group2} =	
...	Should Match Regexp	Bar: 43	(Foo|Bar): (\\d+)

=>

```
${ret} = 'Foo: 42'
${match} = 'Bar: 43'
${group1} = 'Bar'
${group2} = '43'
```

Should Not Be Empty

item, *msg=None*

Verifies that the given item is not empty.

The length of the item is got using the *Get Length* keyword. The default error message can be overridden with the `msg` argument.

Should Not Be Equal

first, *second*, *msg=None*, *values=True*, *ignore_case=False*

Fails if the given objects are equal.

See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`.

If `ignore_case` is given a true value (see *Boolean arguments*) and both arguments are strings, comparison is done case-insensitively. New option in Robot Framework 3.0.1.

Should Not Be Equal As Integers	*first*, *second*, *msg=None*, *values=True*, *base=None*	Fails if objects are equal after converting them to integers. See *Convert To Integer* for information how to convert integers from other bases than 10 using `base` argument or `0b/0o/0x` prefixes. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`. See *Should Be Equal As Integers* for some usage examples.
Should Not Be Equal As Numbers	*first*, *second*, *msg=None*, *values=True*, *precision=6*	Fails if objects are equal after converting them to real numbers. The conversion is done with *Convert To Number* keyword using the given `precision`. See *Should Be Equal As Numbers* for examples on how to use `precision` and why it does not always work as expected. See also *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`.
Should Not Be Equal As Strings	*first*, *second*, *msg=None*, *values=True*, *ignore_case=False*	Fails if objects are equal after converting them to strings. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`. If `ignore_case` is given a true value (see *Boolean arguments*), comparison is done case-insensitively. Strings are always NFC normalized. `ignore_case` is a new feature in Robot Framework 3.0.1.
Should Not Be True	*condition*, *msg=None*	Fails if the given condition is true. See *Should Be True* for details about how `condition` is evaluated and how `msg` can be used to override the default error message.
Should Not Contain	*container*, *item*, *msg=None*, *values=True*, *ignore_case=False*	Fails if `container` contains `item` one or more times. Works with strings, lists, and anything that supports Python's `in` operator. See *Should Be Equal* for an explanation on how to override the default error message with arguments `msg` and `values`. `ignore_case` has exactly the same semantics as with *Should Contain*. Examples: Should Not Contain ${some list} value Should Not Contain ${output} FAILED ignore_case=True
Should Not Contain Any	*container*, **items*, ***configuration*	Fails if `container` contains one or more of the `*items`. Works with strings, lists, and anything that supports Python's `in` operator. Supports additional configuration parameters `msg`, `values` and `ignore_case`, which have exactly the same semantics as arguments with same names have with *Should Contain*. These arguments must always be given using `name=value` syntax after all `items`. Note that possible equal signs in `items` must be escaped with a backslash (e.g. `foo\=bar`) to avoid them to be passed in as `**configuration`. Examples: Should Not Contain Any ${string} substring 1 substring 2 Should Not Contain Any ${list} item 1 item 2 item 3 Should Not Contain Any ${list} item 1 item 2 item 3 ignore_case=True Should Not Contain Any ${list} @{items} msg=Custom message values=False New in Robot Framework 3.0.1.
Should Not End With	*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*	Fails if the string `str1` ends with the string `str2`. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.
Should Not Match	*string*, *pattern*, *msg=None*, *values=True*, *ignore_case=False*	Fails if the given `string` matches the given `pattern`. Pattern matching is similar as matching files in a shell with `*`, `?` and `[chars]` acting as wildcards. See the *Glob patterns* section for more information. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.
Should Not Match Regexp	*string*, *pattern*, *msg=None*, *values=True*	Fails if `string` matches `pattern` as a regular expression. See *Should Match Regexp* for more information about arguments.
Should Not Start With	*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*	Fails if the string `str1` starts with the string `str2`. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.
Should Start With	*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*	Fails if the string `str1` does not start with the string `str2`. See *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`, as well as for semantics of the `ignore_case` option.
Sleep	*time_*, *reason=None*	Pauses the test executed for the given time. `time` may be either a number or a time string. Time strings are in a format such as `1 day 2 hours 3 minutes 4 seconds 5milliseconds` or `1d 2h 3m 4s 5ms`, and they are fully explained in an appendix of Robot Framework User Guide. Optional *reason* can be used to explain why sleeping is necessary. Both the time slept and the reason are logged. Examples: Sleep 42 Sleep 1.5 Sleep 2 minutes 10 seconds Sleep 10s Wait for a reply
Variable Should Exist	*name*, *msg=None*	Fails unless the given variable exists within the current scope. The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*. The default error message can be overridden with the `msg` argument. See also *Variable Should Not Exist* and *Keyword Should Exist*.
Variable Should Not Exist	*name*, *msg=None*	Fails if the given variable exists within the current scope. The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*.

		The default error message can be overridden with the `msg` argument. See also *Variable Should Exist* and *Keyword Should Exist*.
Wait Until Keyword Succeeds	*retry*, *retry_interval*, *name*, **args*	Runs the specified keyword and retries if it fails. `name` and `args` define the keyword that is executed similarly as with *Run Keyword*. How long to retry running the keyword is defined using `retry` argument either as timeout or count. `retry_interval` is the time to wait before trying to run the keyword again after the previous run has failed. If `retry` is given as timeout, it must be in Robot Framework's time format (e.g. `1 minute`, `2 min 3 s`, `4.5`) that is explained in an appendix of Robot Framework User Guide. If it is given as count, it must have `times` or `x` postfix (e.g. `5 times`, `10 x`). `retry_interval` must always be given in Robot Framework's time format. If the keyword does not succeed regardless of retries, this keyword fails. If the executed keyword passes, its return value is returned. Examples: Wait Until Keyword Succeeds | 2 min | 5 sec | My keyword | argument ${result} = | Wait Until Keyword Succeeds | 3x | 200ms | My keyword All normal failures are caught by this keyword. Errors caused by invalid syntax, test or keyword timeouts, or fatal exceptions (caused e.g. by *Fatal Error*) are not caught. Running the same keyword multiple times inside this keyword can create lots of output and considerably increase the size of the generated output files. It is possible to remove unnecessary keywords from the outputs using `--RemoveKeywords WUKS` command line option.

Altogether 105 keywords.
Generated by Libdoc on 2020-05-04 18:02:33.