https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables

Table of Contents

[https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables 1](#_Toc108422875)

[1 Getting started 2](#_Toc108422876)

[**1.1   Introduction** 2](#_Toc108422877)

[**1.1.1   Why Robot Framewor,,,,,,,,,,,k?** 2](#_Toc108422878)

[**1.1.2   High-level architecture** 3](#_Toc108422879)

[**1.1.3   Screenshots** 3](#_Toc108422880)

[**1.1.4   Getting more information** 4](#_Toc108422881)

[**1.2   Copyright and license** 5](#_Toc108422882)

[**1.3   Installation instructions** 5](#_Toc108422883)

[**1.3.1   Python installation** 6](#_Toc108422884)

[**1.3.2   Installing using pip** 8](#_Toc108422885)

[**1.3.3   Installing from source** 10](#_Toc108422886)

[**1.3.4   Verifying installation** 10](#_Toc108422887)

[**1.3.5   Virtual environments** 10](#_Toc108422888)

[**1.4   Demonstrations** 11](#_Toc108422889)

[2   Creating test data 11](#_Toc108422890)

[**2.1   Test data syntax** 11](#_Toc108422891)

[**2.1.1   Files and directories** 12](#_Toc108422892)

[**2.1.2   Test data sections** 12](#_Toc108422893)

[**2.1.3   Supported file formats** 13](#_Toc108422894)

[**2.1.4   Rules for parsing the data** 17](#_Toc108422895)

[**2.2   Creating test cases** 21](#_Toc108422896)

[**2.2.1   Test case syntax** 22](#_Toc108422897)

[**2.2.2   Using arguments** 24](#_Toc108422898)

[**2.2.3   Failures** 29](#_Toc108422899)

[**2.2.4   Test case name and documentation** 30](#_Toc108422900)

[**2.2.5   Tagging test cases** 31](#_Toc108422901)

[**2.2.6   Test setup and teardown** 33](#_Toc108422902)

[**2.2.7   Test templates** 34](#_Toc108422903)

[**2.2.8   Different test case styles** 37](#_Toc108422904)

[**2.3   Creating tasks** 39](#_Toc108422905)

[**2.3.1   Task syntax** 39](#_Toc108422906)

[**2.3.2   Task related settings** 40](#_Toc108422907)

[**2.4   Creating test suites** 40](#_Toc108422908)

[**2.4.1   Test case files** 40](#_Toc108422909)

[**2.4.2   Test suite directories** 40](#_Toc108422910)

[**2.4.3   Test suite name and documentation** 42](#_Toc108422911)

[**2.4.4   Free test suite metadata** 43](#_Toc108422912)

[**2.4.5   Suite setup and teardown** 43](#_Toc108422913)

[**2.5   Using test libraries** 43](#_Toc108422914)

[**2.5.1   Importing libraries** 44](#_Toc108422915)

[**2.5.2   Specifying library to import** 45](#_Toc108422916)

[**2.5.3   Setting custom name to test library** 46](#_Toc108422917)

[**2.5.4   Standard libraries** 46](#_Toc108422918)

[**2.5.5   External libraries** 47](#_Toc108422919)

[**2.6   Variables** 48](#_Toc108422920)

[**2.6.1   Introduction** 48](#_Toc108422921)

[**2.6.2   Using variables** 49](#_Toc108422922)

[**2.6.3   Creating variables** 55](#_Toc108422923)

[**2.6.4   Built-in variables** 61](#_Toc108422924)

[**2.6.5   Variable priorities and scopes** 65](#_Toc108422925)

[**2.6.6   Advanced variable features** 67](#_Toc108422926)

[**2.7   Creating user keywords** 71](#_Toc108422927)

[**2.7.1   User keyword syntax** 72](#_Toc108422928)

[**2.7.2   User keyword name and documentation** 73](#_Toc108422929)

[**2.7.3   User keyword tags** 74](#_Toc108422930)

[**2.7.4   User keyword arguments** 75](#_Toc108422931)

[**2.7.5   Embedding arguments into keyword name** 79](#_Toc108422932)

[**2.7.6   User keyword return values** 83](#_Toc108422933)

[**2.7.7   User keyword teardown** 87](#_Toc108422934)

[**2.8   Resource and variable files** 87](#_Toc108422935)

[**2.8.1   Resource files** 88](#_Toc108422936)

[**2.8.2   Variable files** 89](#_Toc108422937)

[**2.9   Control structures** 97](#_Toc108422938)

[**2.9.1   FOR loops** 98](#_Toc108422939)

[**2.9.2   WHILE loops** 107](#_Toc108422940)

[**2.9.3   Loop control using BREAK and CONTINUE** 109](#_Toc108422941)

[**2.9.4   IF/ELSE syntax** 110](#_Toc108422942)

[**2.9.5   TRY/EXCEPT syntax** 114](#_Toc108422943)

[**2.10   Advanced features** 119](#_Toc108422944)

[**2.10.1   Handling keywords with same names** 119](#_Toc108422945)

[**2.10.2   Timeouts** 121](#_Toc108422946)

[**2.10.3   Parallel execution of keywords** 123](#_Toc108422947)

[3   Executing test cases 123](#_Toc108422948)

[**3.1   Basic usage** 123](#_Toc108422949)

[**3.1.1   Starting test execution** 124](#_Toc108422950)

[**3.1.2   Using command line options** 125](#_Toc108422951)

[**3.1.3   Test results** 128](#_Toc108422952)

[**3.1.4   Argument files** 130](#_Toc108422953)

[**3.1.5   Getting help and version information** 131](#_Toc108422954)

[**3.1.6   Creating start-up scripts** 132](#_Toc108422955)

[**3.1.7   Making \*.robot files executable** 133](#_Toc108422956)

[**3.1.8   Debugging problems** 134](#_Toc108422957)

[**3.2   Test execution** 135](#_Toc108422958)

[**3.2.1   Execution flow** 135](#_Toc108422959)

[**3.2.2   Test and suite statuses** 137](#_Toc108422960)

[**3.2.3   Continue on failure** 140](#_Toc108422961)

[**3.2.4   Stopping test execution gracefully** 143](#_Toc108422962)

[**3.3   Task execution** 144](#_Toc108422963)

[**3.3.1   Generic automation mode** 144](#_Toc108422964)

[**3.3.2   Task related command line options** 145](#_Toc108422965)

[**3.4   Post-processing outputs** 145](#_Toc108422966)

[**3.4.1   Using Rebot** 145](#_Toc108422967)

[**3.4.2   Creating different reports and logs** 146](#_Toc108422968)

[**3.4.3   Combining outputs** 146](#_Toc108422969)

[**3.4.4   Merging outputs** 147](#_Toc108422970)

[**3.5   Configuring execution** 148](#_Toc108422971)

[**3.5.1   Selecting files to parse** 149](#_Toc108422972)

[**3.5.2   Selecting test cases** 149](#_Toc108422973)

[**3.5.3   Setting metadata** 152](#_Toc108422974)

[**3.5.4   Configuring where to search libraries and other extensions** 153](#_Toc108422975)

[**3.5.5   Setting variables** 155](#_Toc108422976)

[**3.5.6   Dry run** 155](#_Toc108422977)

[**3.5.7   Randomizing execution order** 156](#_Toc108422978)

[**3.5.8   Programmatic modification of test data** 156](#_Toc108422979)

[**3.5.9   Controlling console output** 160](#_Toc108422980)

[**3.5.10   Setting listeners** 162](#_Toc108422981)

[**3.6   Created outputs** 162](#_Toc108422982)

[**3.6.1   Different output files** 163](#_Toc108422983)

[**3.6.2   Log levels** 169](#_Toc108422984)

[**3.6.3   Splitting logs** 170](#_Toc108422985)

[**3.6.4   Configuring statistics** 171](#_Toc108422986)

[**3.6.5   Removing and flattening keywords** 173](#_Toc108422987)

[**3.6.6   Automatically expanding keywords** 175](#_Toc108422988)

[**3.6.7   Setting start and end time of execution** 175](#_Toc108422989)

[**3.6.8   Limiting error message length in reports** 176](#_Toc108422990)

[**3.6.9   Programmatic modification of results** 176](#_Toc108422991)

[**3.6.10   System log** 177](#_Toc108422992)

[4   Extending Robot Framework 178](#_Toc108422993)

[**4.1   Creating test libraries** 178](#_Toc108422994)

[**4.1.1   Introduction** 179](#_Toc108422995)

[**4.1.2   Creating test library class or module** 180](#_Toc108422996)

[**4.1.3   Creating keywords** 185](#_Toc108422997)

[**4.1.4   Communicating with Robot Framework** 209](#_Toc108422998)

[**4.1.5   Distributing test libraries** 218](#_Toc108422999)

[**4.1.6   Dynamic library API** 221](#_Toc108423000)

[**4.1.7   Hybrid library API** 229](#_Toc108423001)

[**4.1.8   Using Robot Framework's internal modules** 231](#_Toc108423002)

[**4.1.9   Extending existing test libraries** 232](#_Toc108423003)

[**4.2   Remote library interface** 234](#_Toc108423004)

[**4.2.1   Introduction** 234](#_Toc108423005)

[**4.2.2   Putting Remote library to use** 235](#_Toc108423006)

[**4.2.3   Supported argument and return value types** 236](#_Toc108423007)

[**4.2.4   Remote protocol** 237](#_Toc108423008)

[**4.3   Listener interface** 241](#_Toc108423009)

[**4.3.1   Taking listeners into use** 242](#_Toc108423010)

[**4.3.2   Listener interface versions** 243](#_Toc108423011)

[**4.3.3   Listener interface methods** 243](#_Toc108423012)

[**4.3.4   Listeners logging** 249](#_Toc108423013)

[**4.3.5   Listener examples** 250](#_Toc108423014)

[**4.3.6   Libraries as listeners** 253](#_Toc108423015)

[5   Supporting Tools 254](#_Toc108423016)

[**5.1   Library documentation tool (Libdoc)** 255](#_Toc108423017)

[**5.1.1   General usage** 255](#_Toc108423018)

[**5.1.2   Writing documentation** 260](#_Toc108423019)

[**5.1.3   Documentation syntax** 263](#_Toc108423020)

[**5.1.4   Internal linking** 266](#_Toc108423021)

[**5.1.5   Representing arguments** 268](#_Toc108423022)

[**5.1.6   Libdoc example** 269](#_Toc108423023)

[**5.2   Test data documentation tool (Testdoc)** 270](#_Toc108423024)

[**5.2.1   General usage** 271](#_Toc108423025)

[**5.2.2   Generating documentation** 272](#_Toc108423026)

[**5.3   Test data clean-up tool (Tidy)** 272](#_Toc108423027)

[**5.4   External tools** 272](#_Toc108423028)

[6   Appendices 273](#_Toc108423029)

[**6.1   All available settings in test data** 273](#_Toc108423030)

[**6.1.1   Setting section** 273](#_Toc108423031)

[**6.1.2   Test Case section** 274](#_Toc108423032)

[**6.1.3   Keyword section** 274](#_Toc108423033)

[**6.2   All command line options** 275](#_Toc108423034)

[**6.2.1   Command line options for test execution** 275](#_Toc108423035)

[**6.2.2   Command line options for post-processing outputs** 279](#_Toc108423036)

[**6.2.3   Environment variables for execution and post-processing** 283](#_Toc108423037)

[**6.3   Documentation formatting** 283](#_Toc108423038)

[**6.3.1   Representing newlines** 284](#_Toc108423039)

[**6.3.2   Paragraphs** 285](#_Toc108423040)

[**6.3.3   Inline styles** 285](#_Toc108423041)

[**6.3.4   URLs** 286](#_Toc108423042)

[**6.3.5   Custom links and images** 286](#_Toc108423043)

[**6.3.6   Section titles** 287](#_Toc108423044)

[**6.3.7   Tables** 288](#_Toc108423045)

[**6.3.8   Lists** 288](#_Toc108423046)

[**6.3.9   Preformatted text** 288](#_Toc108423047)

[**6.3.10   Horizontal ruler** 288](#_Toc108423048)

[**6.4   Time format** 288](#_Toc108423049)

[**6.4.1   Time as number** 288](#_Toc108423050)

[**6.4.2   Time as time string** 288](#_Toc108423051)

[**6.4.3   Time as "timer" string** 288](#_Toc108423052)

[**6.5   Boolean arguments** 288](#_Toc108423053)

[**6.6   Evaluating expressions** 288](#_Toc108423054)

[**6.6.1   Evaluation namespace** 288](#_Toc108423055)

[**6.6.2   Using variables** 288](#_Toc108423056)

[**6.7   Internal API** 288](#_Toc108423057)

**1 Gettin[g started](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "toc-entry-1)**

* [1.1   Introduction](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#introduction)
* [1.2   Copyright and license](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#copyright-and-license)
* [1.3   Installation instructions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installation-instructions)
* [1.4   Demonstrations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#demonstrations)

[**1.1   Introduction**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-183)

Robot Framework is a Python-based, extensible keyword-driven automation framework for acceptance testing, acceptance test driven development (ATDD), behavior driven development (BDD) and robotic process automation (RPA). It can be used in distributed, heterogeneous environments, where automation requires using different technologies and interfaces.

The framework has a rich ecosystem around it consisting of various generic libraries and tools that are developed as separate projects. For more information about Robot Framework and the ecosystem, see [http://robotframework.org](http://robotframework.org/).

Robot Framework is open source software released under the [Apache License 2.0](http://apache.org/licenses/LICENSE-2.0). Its development is sponsored by the [Robot Framework Foundation](http://robotframework.org/foundation).

**Note**

The official RPA support was added in Robot Framework 3.1. This User Guide still talks mainly about creating tests, test data, and test libraries, but same concepts apply also when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks).

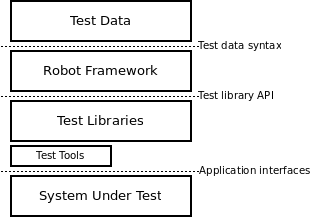
* [1.1.1   Why Robot Framework?](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#why-robot-framework)
* [1.1.2   High-level architecture](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#high-level-architecture)
* [1.1.3   Screenshots](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#screenshots)
* [1.1.4   Getting more information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-more-information)
  + [Project pages](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#project-pages)
  + [Mailing lists](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#mailing-lists)

[**1.1.1   Why Robot Framework?**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-187)

* Enables easy-to-use tabular syntax for [creating test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-cases) in a uniform way.
* Provides ability to create reusable [higher-level keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) from the existing keywords.
* Provides easy-to-read result [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) and [logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) in HTML format.
* Is platform and application independent.
* Provides a simple [library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) for creating customized test libraries which can be implemented natively with Python.
* Provides a [command line interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-test-cases-1) and XML based [output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) for integration into existing build infrastructure (continuous integration systems).
* Provides support for testing web applications, rest APIs, mobile applications, running processes, connecting to remote systems via Telnet or SSH, and so on.
* Supports creating [data-driven test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style).
* Has built-in support for [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables), practical particularly for testing in different environments.
* Provides [tagging](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases) to categorize and [select test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases) to be executed.
* Enables easy integration with source control: [test suites](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite) are just files and directories that can be versioned with the production code.
* Provides [test-case](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) and [test-suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) -level setup and teardown.
* The modular architecture supports creating tests even for applications with several diverse interfaces.

[**1.1.2   High-level architecture**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-188)

Robot Framework is a generic, application and technology independent framework. It has a highly modular architecture illustrated in the diagram below.



*Robot Framework architecture*

The [test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-data) is in simple, easy-to-edit tabular format. When Robot Framework is started, it processes the data, [executes test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-test-cases-1) and generates logs and reports. The core framework does not know anything about the target under test, and the interaction with it is handled by [libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries). Libraries can either use application interfaces directly or use lower level test tools as drivers.

[**1.1.3   Screenshots**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-189)

Following screenshots show examples of the [test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-data) and created [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) and [logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file).

텍스트이(가) 표시된 사진

자동 생성된 설명

*Test case file*

텍스트이(가) 표시된 사진

자동 생성된 설명

*Reports and logs*

[**1.1.4   Getting more information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-190)

[**Project pages**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-191)

The number one place to find more information about Robot Framework and the rich ecosystem around it is [http://robotframework.org](http://robotframework.org/). Robot Framework itself is hosted on [GitHub](https://github.com/robotframework/robotframework).

[**Mailing lists**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-192)

There are several Robot Framework mailing lists where to ask and search for more information. The mailing list archives are open for everyone (including the search engines) and everyone can also join these lists freely. Only list members can send mails, though, and to prevent spam new users are moderated which means that it might take a little time before your first message goes through. Do not be afraid to send question to mailing lists but remember [How To Ask Questions The Smart Way](http://www.catb.org/~esr/faqs/smart-questions.html).

[robotframework-users](http://groups.google.com/group/robotframework-users)

General discussion about all Robot Framework related issues. Questions and problems can be sent to this list. Used also for information sharing for all users.

[robotframework-announce](http://groups.google.com/group/robotframework-announce)

An announcements-only mailing list where only moderators can send messages. All announcements are sent also to the robotframework-users mailing list so there is no need to join both lists.

[robotframework-devel](http://groups.google.com/group/robotframework-devel)

Discussion about Robot Framework development.

[**1.2   Copyright and license**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-184)

Robot Framework is open source software provided under the [Apache License 2.0](http://apache.org/licenses/LICENSE-2.0). Robot Framework documentation such as this User Guide use the [Creative Commons Attribution 3.0 Unported](http://creativecommons.org/licenses/by/3.0) license. Most libraries and tools in the larger ecosystem around the framework are also open source, but they may use different licenses.

The full Robot Framework copyright notice is included below:

Copyright 2008-2015 Nokia Networks

Copyright 2016- Robot Framework Foundation

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

[**1.3   Installation instructions**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-185)

These instructions cover installing [Robot Framework](https://robotframework.org/) and its preconditions on different operating systems. If you already have [Python](http://python.org/) installed, you can install Robot Framework using the standard package manager [pip](https://pip.pypa.io/):

pip install robotframework

* [1.3.1   Python installation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#python-installation)
  + [Installing Python on Linux](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-python-on-linux)
  + [Installing Python on Windows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-python-on-windows)
  + [Installing Python on macOS](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-python-on-macos)
  + [PyPy installation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pypy-installation)
  + [Configuring PATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-path)
* [1.3.2   Installing using pip](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-using-pip)
  + [Running pip command](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-pip-command)
  + [Installing and uninstalling Robot Framework](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-and-uninstalling-robot-framework)
* [1.3.3   Installing from source](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installing-from-source)
* [1.3.4   Verifying installation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#verifying-installation)
* [1.3.5   Virtual environments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#virtual-environments)

[**1.3.1   Python installation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-193)

[Robot Framework](https://robotframework.org/) is implemented using [Python](http://python.org/), and a precondition to install it is having Python or its alternative implementation [PyPy](https://pypy.org/) installed. Another recommended precondition is having the [pip](https://pip.pypa.io/) package manager available.

Robot Framework requires Python 3.6 or newer. If you need to use Python 2, [Jython](http://jython.org/) or [IronPython](http://ironpython.net/), you can use [Robot Framework 4.1.3](https://github.com/robotframework/robotframework/tree/v4.1.3#readme).

[**Installing Python on Linux**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-194)

On Linux you should have suitable Python installation with [pip](https://pip.pypa.io/) available by default. If not, you need to consult your distributions documentation to learn how to install them. This is also true if you want to use some other Python version than the one provided by your distribution by default.

To check what Python version you have installed, you can run python --version command in a terminal:

$ python --version

Python 3.8.10

Notice that if your distribution provides also older Python 2, running python may use that. To use Python 3, you can use python3 command or even more version specific command like python3.8. You need to use these version specific variants also if you have multiple Python 3 versions installed and need to pinpoint which one to use:

$ python3.9 --version

Python 3.9.10

$ python3.10 --version

Python 3.10.2

Installing Robot Framework directly under the system provided Python has a risk that possible problems can affect the whole Python installation used also by the operating system itself. Nowadays Linux distributions typically use [user installs](https://pip.pypa.io/en/stable/user_guide/#user-installs) by default to avoid such problems, but users can also themselves decide to use [virtual environments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#virtual-environments).

[**Installing Python on Windows**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-195)

On Windows Python is not available by default, but it is easy to install. The recommended way to install it is using the official Windows installers available at [http://python.org](http://python.org/). For other alternatives, such as installing from the Microsoft Store, see the [official Python documentation](https://docs.python.org/3/using/windows.html).

When installing Python on Windows, it is recommended to add Python to [PATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-path) to make it and tools like pip and Robot Framework easier to execute from the command line. When using the [official installer](https://docs.python.org/3/using/windows.html#windows-full), you just need to select the Add Python 3.x to PATH checkbox on the first dialog.

To make sure Python installation has been successful and Python has been added to PATH, you can open the command prompt and execute python --version:

C:\*>python --version*

Python 3.9.4

If you install multiple Python versions on Windows, the Python that is used when you execute python is the one first in PATH. If you need to use others, the easiest way is using the [py launcher](https://docs.python.org/3/using/windows.html" \l "launcher):

C:\*>py --version*

Python 3.9.4

C:\*>py -3.10 --version*

Python 3.10.2

[**Installing Python on macOS**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-196)

MacOS does not provide Python 3 compatible Python version by default, so it needs to be installed separately. The recommended approach is using the official macOS installers available at [http://python.org](http://python.org/). If you are using a package manager like [Homebrew](https://brew.sh/), installing Python via it is possible as well.

You can validate Python installation on macOS using python --version like on other operating systems.

[**PyPy installation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-197)

[PyPy](https://pypy.org/) is an alternative Python implementation. Its main advantage over the standard Python implementation is that it can be faster and use less memory, but this depends on the context where and how it is used. If execution speed is important, at least testing PyPy is probably a good idea.

Installing PyPy is a straightforward procedure and you can find both installers and installation instructions at [http://pypy.org](http://pypy.org/). To validate that PyPy installation was successful, run pypy --version or pypy3 --version.

**Note**

Using Robot Framework with PyPy is officially supported only on Linux.

[**Configuring PATH**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-198)

The [PATH environment variable](https://en.wikipedia.org/wiki/PATH_(variable)) lists directories where commands executed in a system are searched from. To make using Python, [pip](https://pip.pypa.io/) and Robot Framework easier from the command line, it is recommended to add the Python installation directory as well as the directory where commands like pip and robot are installed into PATH.

When using Python on Linux or macOS, Python and tools installed with it should be automatically in PATH. If you nevertheless need to update PATH, you typically need to edit some system wide or user specific configuration file. Which file to edit and how depends on the operating system and you need to consult its documentation for more details.

On Windows the easiest way to make sure PATH is configured correctly is setting the Add Python 3.x to PATH checkbox when [running the installer](https://docs.python.org/3/using/windows.html#the-full-installer). To manually modify PATH on Windows, follow these steps:

1. Find Environment Variables under Settings. There are variables affecting the whole system and variables affecting only the current user. Modifying the former will require admin rights, but modifying the latter is typically enough.
2. Select PATH (often written like Path) and click Edit. If you are editing user variables and PATH does not exist, click New instead.
3. Add both the Python installation directory and the *Scripts* directory under the installation directory into PATH.
4. Exit the dialog with Ok to save the changes.
5. Start a new command prompt for the changes to take effect.

[**1.3.2   Installing using pip**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-199)

These instructions cover installing Robot Framework using [pip](https://pip.pypa.io/), the standard Python package manager. If you are using some other package manager like [Conda](https://conda.io/), you can use it instead but need to study its documentation for instructions.

When installing Python, you typically get pip installed automatically. If that is not the case, you need to check the documentation of that Python installation for instructions how to install it separately.

[**Running pip command**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-200)

Typically you use pip by running the pip command, but on Linux you may need to use pip3 or even more Python version specific variant like pip3.8 instead. When running pip or any of its variants, the pip version that is found first in [PATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-path) will be used. If you have multiple Python versions installed, you may need to pinpoint which exact version you want to use. This is typically easiest done by running python -m pip and substituting python with the Python version you want to use.

To make sure you have pip available, you can run pip --version or equivalent.

Examples on Linux:

$ pip --version

pip 20.2 from ... (python 3.8)

$ python3.10 -m pip --version

pip 20.2 from ... (python 3.10)

Examples on Windows:

C:\*> pip --version*

pip 20.2.2 from ... (python 3.9)

C:\*> py -m 3.10 -m pip --version*

pip 20.2.1 from ... (python 3.10)

In the subsequent sections pip is always run using the pip command. You may need to use some of the other approaches explained above in your environment.

[**Installing and uninstalling Robot Framework**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-201)

The easiest way to use pip is by letting it find and download packages it installs from the [Python Package Index (PyPI)](https://pypi.org/project/robotframework), but it can also install packages downloaded from the PyPI separately. The most common usages are shown below and [pip](https://pip.pypa.io/) documentation has more information and examples.

*# Install the latest version (does not upgrade)*

pip install robotframework

*# Upgrade to the latest stable version*

pip install --upgrade robotframework

*# Upgrade to the latest version even if it is a pre-release*

pip install --upgrade --pre robotframework

*# Install a specific version*

pip install robotframework==5.0

*# Install separately downloaded package (no network connection needed)*

pip install robotframework-5.0-py3-none-any.whl

*# Install latest (possibly unreleased) code directly from GitHub*

pip install https://github.com/robotframework/robotframework/archive/master.zip

*# Uninstall*

pip uninstall robotframework

[**1.3.3   Installing from source**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-202)

Another installation alternative is getting Robot Framework source code and installing it using the provided setup.py script. This approach is recommended only if you do not have [pip](https://pip.pypa.io/) available for some reason.

You can get the source code by downloading a source distribution as a zip package from [PyPI](https://pypi.org/project/robotframework) and extracting it. An alternative is cloning the [GitHub](https://github.com/robotframework/robotframework) repository and checking out the needed release tag.

Once you have the source code, you can install it with the following command:

python setup.py install

The setup.py script accepts several arguments allowing, for example, installation into a non-default location that does not require administrative rights. It is also used for creating different distribution packages. Run python setup.py --help for more details.

[**1.3.4   Verifying installation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-203)

To make sure that the correct Robot Framework version has been installed, run the following command:

$ robot --version

Robot Framework 5.0 (Python 3.8.10 on linux)

If running these commands fails with a message saying that the command is not found or recognized, a good first step is double-checking the [PATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-path) configuration.

If you have installed Robot Framework under multiple Python versions, running robot will execute the one first in [PATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-path). To select explicitly, you can run python -m robot and substitute python with the right Python version.

$ python3.10 -m robot --version

Robot Framework 5.0 (Python 3.10.2 on linux)

C:**\>**py -3.10 -m robot --version

Robot Framework 5.0 (Python 3.10.3 on win32)

[**1.3.5   Virtual environments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-204)

Python [virtual environments](https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/#creating-a-virtual-environment) allow Python packages to be installed in an isolated location for a particular system or application, rather than installing all packages into the same global location. They have two main use cases:

* Install packages needed by different projects into their own environments. This avoids conflicts if projects need different versions of same packages.
* Avoid installing everything under the global Python installation. This is especially important on Linux where the global Python installation may be used by the distribution itself and messing it up can cause severe problems.

[**1.4   Demonstrations**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-186)

There are several demo projects that introduce Robot Framework and help getting started with it.

[Quick Start Guide](https://github.com/robotframework/QuickStartGuide/blob/master/QuickStart.rst)

Introduces the most important features of Robot Framework and acts as an executable demo.

[Robot Framework demo](https://github.com/robotframework/RobotDemo)

Simple example test cases. Demonstrates also creating custom test libraries.

[Web testing demo](https://github.com/robotframework/WebDemo)

Demonstrates how to create tests and higher level keywords. The system under test is a simple web page that is tested using [SeleniumLibrary](https://github.com/robotframework/SeleniumLibrary).

[ATDD with Robot Framework](https://code.google.com/p/atdd-with-robot-framework)

Demonstrates how to use Robot Framework when following Acceptance Test Driven Development (ATDD) process.

[**2   Creating test data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-15)

* [2.1   Test data syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-syntax)
* [2.2   Creating test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-cases)
* [2.3   Creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks)
* [2.4   Creating test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-suites)
* [2.5   Using test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries)
* [2.6   Variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables)
* [2.7   Creating user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords)
* [2.8   Resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files)
* [2.9   Control structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#control-structures)
* [2.10   Advanced features](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#advanced-features)

[**2.1   Test data syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-205)

This section covers Robot Framework's overall test data syntax. The following sections will explain how to actually create test cases, test suites and so on. Although this section mostly uses term *test*, the same rules apply also when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks).

* [2.1.1   Files and directories](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#files-and-directories)
* [2.1.2   Test data sections](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-sections)
* [2.1.3   Supported file formats](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats)
  + [Space separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-separated-format)
  + [Pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format)
  + [reStructuredText format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#restructuredtext-format)
* [2.1.4   Rules for parsing the data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#rules-for-parsing-the-data)
  + [Ignored data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#ignored-data)
  + [Escaping](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping)
  + [Dividing data to several rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows)

[**2.1.1   Files and directories**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-215)

The hierarchical structure for arranging test cases is built as follows:

* Test cases are created in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files).
* A test case file automatically creates a [test suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-suites) containing the test cases in that file.
* A directory containing test case files forms a higher-level test suite. Such a [test suite directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories) has suites created from test case files as its child test suites.
* A test suite directory can also contain other test suite directories, and this hierarchical structure can be as deeply nested as needed.
* Test suite directories can have a special [initialization file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files) configuring the created test suite.

In addition to this, there are:

* [Test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) containing the lowest-level keywords.
* [Resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) with [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) and higher-level [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords).
* [Variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files) to provide more flexible ways to create variables than resource files.

Test case files, test suite initialization files and resource files are all created using Robot Framework test data syntax. Test libraries and variable files are created using "real" programming languages, most often Python.

[**2.1.2   Test data sections**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-216)

Robot Framework data is defined in different sections, often also called tables, listed below:

| *Different sections in data* | |
| --- | --- |
| **Section** | **Used for** |
| Settings | 1) Importing [test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries), [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) and [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files).  2) Defining metadata for [test suites](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite) and [test cases](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestCase). |
| Variables | Defining [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) that can be used elsewhere in the test data. |
| Test Cases | [Creating test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-cases) from available keywords. |
| Tasks | [Creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks) using available keywords. Single file can only contain either tests or tasks. |
| Keywords | [Creating user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) from existing lower-level keywords |
| Comments | Additional comments or data. Ignored by Robot Framework. |

Different sections are recognized by their header row. The recommended header format is \*\*\* Settings \*\*\*, but the header is case-insensitive, surrounding spaces are optional, and the number of asterisk characters can vary as long as there is one asterisk in the beginning. In addition to using the plural format, also singular variants like Setting and Test Case are accepted. In other words, also \*setting would be recognized as a section header.

The header row can contain also other data than the actual section header. The extra data must be separated from the section header using the data format dependent separator, typically two or more spaces. These extra headers are ignored at parsing time, but they can be used for documenting purposes. This is especially useful when creating test cases using the [data-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style).

Possible data before the first section is ignored.

**Note**

Section names used to be space-insensitive, but that was deprecated in Robot Framework 3.1 and trying to use something like TestCases or S e t t i n g s causes an error in Robot Framework 3.2.

**Note**

Prior to Robot Framework 3.1, all unrecognized sections were silently ignored but nowadays they cause an error. Comments sections can be used if sections not containing actual test data are needed.

[**2.1.3   Supported file formats**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-217)

The most common approach to create Robot Framework data is using the [space separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-separated-format) where pieces of the data, such as keywords and their arguments, are separated from each others with two or more spaces. An alternative is using the [pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format) where the separator is the pipe character surrounded with spaces (|).

Executed files typically use the *.robot* extension, but that [can be configured](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse) with the --extension option. [Resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) can use the *.robot* extension as well, but using the dedicated *.resource* extension is recommended. Files containing non-ASCII characters must be saved using the UTF-8 encoding.

Robot Framework also supports [reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText) files so that normal Robot Framework data is [embedded into code blocks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#restructuredtext-format). It is possible to use either *.rst* or *.rest* extension with reStructuredText files, but the aforementioned --extension option [must be used](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse) to enable parsing them when executing a directory.

Earlier Robot Framework versions supported data also in HTML and TSV formats. The TSV format still works if the data is compatible with the [space separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-separated-format), but the support for the HTML format has been removed altogether. If you encounter such data files, you need to convert them to the plain text format to be able to use them with Robot Framework 3.2 or newer. The easiest way to do that is using the [Tidy](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tidy) tool, but you must use the version included with Robot Framework 3.1 because newer versions do not understand the HTML format at all.

[**Space separated format**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-218)

When Robot Framework parses data, it first splits the data to lines and then lines to tokens such as keywords and arguments. When using the space separated format, the separator between tokens is two or more spaces or alternatively one or more tab characters. In addition to the normal ASCII space, any Unicode character considered to be a space (e.g. no-break space) works as a separator. The number of spaces used as separator can vary, as long as there are at least two, making it possible to align the data nicely in settings and elsewhere when it makes the data easier to understand.

**\*\*\* Settings \*\*\***

**Documentation** Example using the space separated format.

**Library** **OperatingSystem**

**\*\*\* Variables \*\*\***

${MESSAGE} Hello, world!

**\*\*\* Test Cases \*\*\***

**My Test**

[**Documentation**] Example test.

Log ${MESSAGE}

My Keyword ${CURDIR}

**Another Test**

Should Be Equal ${MESSAGE} Hello, world!

**\*\*\* Keywords \*\*\***

**My Keyword**

[**Arguments**] ${path}

Directory Should Exist ${path}

Because tabs and consecutive spaces are considered separators, they must be escaped if they are needed in keyword arguments or elsewhere in the actual data. It is possible to use special escape syntax like \t for tab and \xA0 for no-break space as well as [built-in variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables) ${SPACE} and ${EMPTY}. See the [Escaping](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) section for details.

**Tip**

Although using two spaces as a separator is enough, it is recommended to use four spaces to make the separator easier to recognize.

**Note**

Prior to Robot Framework 3.2, non-ASCII spaces used in the data were converted to ASCII spaces during parsing. Nowadays all data is preserved as-is.

[**Pipe separated format**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-219)

The biggest problem of the space delimited format is that visually separating keywords from arguments can be tricky. This is a problem especially if keywords take a lot of arguments and/or arguments contain spaces. In such cases the pipe delimited variant can work better because it makes the separator more visible.

One file can contain both space separated and pipe separated lines. Pipe separated lines are recognized by the mandatory leading pipe character, but the pipe at the end of the line is optional. There must always be at least one space or tab on both sides of the pipe except at the beginning and at the end of the line. There is no need to align the pipes, but that often makes the data easier to read.

| **\*\*\* Settings \*\*\*** |

| **Documentation** | Example using the pipe separated format.

| **Library** | **OperatingSystem**

| **\*\*\* Variables \*\*\*** |

| ${MESSAGE} | Hello, world!

| **\*\*\* Test Cases \*\*\*** | | |

| **My Test** | [**Documentation**] | Example test. |

| | Log | ${MESSAGE} |

| | My Keyword | ${CURDIR} |

| **Another Test** | Should Be Equal | ${MESSAGE} | Hello, world!

| **\*\*\* Keywords \*\*\*** | | |

| **My Keyword** | [**Arguments**] | ${path} |

| | Directory Should Exist | ${path} |

When using the pipe separated format, consecutive spaces or tabs inside arguments do not need to be escaped. Similarly empty columns do not need to be escaped except [if they are at the end](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping). Possible pipes surrounded by spaces in the actual test data must be escaped with a backslash, though:

| **\*\*\* Test Cases \*\*\*** | | | |

| **Escaping Pipe** | ${file count} = | Execute Command | ls -1 \*.txt \| wc -l |

| | Should Be Equal | ${file count} | 42 |

**Note**

Preserving consecutive spaces and tabs in arguments is new in Robot Framework 3.2. Prior to it non-ASCII spaces used in the data were also converted to ASCII spaces.

[**reStructuredText format**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-220)

[reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText) (reST) is an easy-to-read plain text markup syntax that is commonly used for documentation of Python projects, including Python itself as well as this User Guide. reST documents are most often compiled to HTML, but also other output formats are supported. Using reST with Robot Framework allows you to mix richly formatted documents and test data in a concise text format that is easy to work with using simple text editors, diff tools, and source control systems.

**Note**

Using [reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText) files with Robot Framework requires the Python [docutils](https://pypi.python.org/pypi/docutils) module to be installed.

When using Robot Framework with reStructuredText files, normal Robot Framework data is embedded to so called code blocks. In standard reST, code blocks are marked using the ‘code’ directive, but Robot Framework supports also ‘code-block’ or ‘sourcecode’ directives used by the [Sphinx](http://sphinx-doc.org/) tool.

**reStructuredText example**

**------------------------**

This text is outside code blocks and thus ignored.

.. **code**:: **robotframework**

**\*\*\* Settings \*\*\***

**Documentation** Example using the reStructuredText format.

**Library** **OperatingSystem**

**\*\*\* Variables \*\*\***

${MESSAGE} Hello, world!

**\*\*\* Test Cases \*\*\***

**My Test**

[**Documentation**] Example test.

Log ${MESSAGE}

My Keyword ${CURDIR}

**Another Test**

Should Be Equal ${MESSAGE} Hello, world!

Also, this text is outside code blocks and ignored. Code blocks not

containing Robot Framework data are ignored as well.

.. **code**:: **robotframework**

*# Both space and pipe separated formats are supported.*

| **\*\*\* Keyword \*\*\*** | | |

| **My Keyword** | [**Arguments**] | ${path} |

| | Directory Should Exist | ${path} |

.. **code**:: **python**

*# This code block is ignored.*

**def** example():

print('Hello, world!')

Robot Framework supports reStructuredText files using both *.rst* and *.rest* extension. When executing a directory containing reStructuredText files, the --extension option must be used to explicitly tell that [these files should be parsed](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse).

When Robot Framework parses reStructuredText files, errors below level SEVERE are ignored to avoid noise about possible non-standard directives and other such markup. This may hide also real errors, but they can be seen when processing files using reStructuredText tooling normally.

[**2.1.4   Rules for parsing the data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-221)

[**Ignored data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-222)

When Robot Framework parses the test data files, it ignores:

* All data before the first [test data section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-sections).
* Data in the [Comments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-sections) section.
* All empty rows.
* All empty cells at the end of rows when using the [pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format).
* All single backslashes (\) when not used for [escaping](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping).
* All characters following the hash character (#), when it is the first character of a cell. This means that hash marks can be used to enter comments in the test data.

When Robot Framework ignores some data, this data is not available in any resulting reports and, additionally, most tools used with Robot Framework also ignore them. To add information that is visible in Robot Framework outputs, place it to the documentation or other metadata of test cases or suites, or log it with the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Log* or *Comment*.

[**Escaping**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-223)

The escape character in Robot Framework test data is the backslash (\) and additionally [built-in variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables) ${EMPTY} and ${SPACE} can often be used for escaping. Different escaping mechanisms are discussed in the sections below.

**Escaping special characters**

The backslash character can be used to escape special characters so that their literal values are used.

| *Escaping special characters* | | |
| --- | --- | --- |
| **Character** | **Meaning** | **Examples** |
| \$ | Dollar sign, never starts a [scalar variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variable). | \${notvar} |
| \@ | At sign, never starts a [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable). | \@{notvar} |
| \& | Ampersand, never starts a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable). | \&{notvar} |
| \% | Percent sign, never starts an [environment variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#environment-variable). | \%{notvar} |
| \# | Hash sign, never starts a [comment](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#comment). | \# not comment |
| \= | Equal sign, never part of [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax). | not\=named |
| \| | Pipe character, not a separator in the [pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format). | ls -1 \*.txt \| wc -l |
| \\ | Backslash character, never escapes anything. | c:\\temp, \\${var} |

**Forming escape sequences**

The backslash character also allows creating special escape sequences that are recognized as characters that would otherwise be hard or impossible to create in the test data.

| *Escape sequences* | | |
| --- | --- | --- |
| **Sequence** | **Meaning** | **Examples** |
| \n | Newline character. | first line\n2nd line |
| \r | Carriage return character | text\rmore text |
| \t | Tab character. | text\tmore text |
| \xhh | Character with hex value hh. | null byte: \x00, ä: \xE4 |
| \uhhhh | Character with hex value hhhh. | snowman: \u2603 |
| \Uhhhhhhhh | Character with hex value hhhhhhhh. | love hotel: \U0001f3e9 |

**Note**

All strings created in the test data, including characters like \x02, are Unicode and must be explicitly converted to byte strings if needed. This can be done, for example, using *Convert To Bytes* or *Encode String To Bytes* keywords in [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) and [String](https://robotframework.org/robotframework/latest/libraries/String.html) libraries, respectively, or with something like value.encode('UTF-8') in Python code.

**Note**

If invalid hexadecimal values are used with \x, \u or \U escapes, the end result is the original value without the backslash character. For example, \xAX (not hex) and \U00110000 (too large value) result with xAX and U00110000, respectively. This behavior may change in the future, though.

**Note**

[Built-in variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variable) ${\n} can be used if operating system dependent line terminator is needed (\r\n on Windows and \n elsewhere).

**Handling empty values**

When using the [space separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-separated-format), the number of spaces used as a separator can vary and thus empty values cannot be recognized unless they are escaped. Empty cells can be escaped either with the backslash character or with [built-in variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variable) ${EMPTY}. The latter is typically recommended as it is easier to understand.

**\*\*\* Test Cases \*\*\***

**Using backslash**

Do Something first arg \

Do Something \ second arg

**Using** ${EMPTY}

Do Something first arg ${EMPTY}

Do Something ${EMPTY} second arg

When using the [pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format), empty values need to be escaped only when they are at the end of the row:

| **\*\*\* Test Cases \*\*\*** | | | |

| **Using backslash** | Do Something | first arg | \ |

| | Do Something | | second arg |

| | | | |

| **Using** ${EMPTY} | Do Something | first arg | ${EMPTY} |

| | Do Something | | second arg |

**Handling spaces**

Spaces, especially consecutive spaces, as part of arguments for keywords or needed otherwise are problematic for two reasons:

* Two or more consecutive spaces is considered a separator when using the [space separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-separated-format).
* Leading and trailing spaces are ignored when using the [pipe separated format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pipe-separated-format).

In these cases, spaces need to be escaped. Similarly as when escaping empty values, it is possible to do that either by using the backslash character or by using the [built-in variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variable) ${SPACE}.

| *Escaping spaces examples* | | |
| --- | --- | --- |
| **Escaping with backslash** | **Escaping with ${SPACE}** | **Notes** |
| \ leading space | ${SPACE}leading space |  |
| trailing space \ | trailing space${SPACE} | Backslash must be after the space. |
| \ \ | ${SPACE} | Backslash needed on both sides. |
| consecutive \ \ spaces | consecutive${SPACE \* 3}spaces | Using [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax). |

As the above examples show, using the ${SPACE} variable often makes the test data easier to understand. It is especially handy in combination with the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) when more than one space is needed.

[**Dividing data to several rows**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-224)

If there is more data than readily fits a row, it is possible to split it and start continuing rows with ellipsis (...). Ellipses can be indented to match the indentation of the starting row and they must always be followed by the normal test data separator.

In most places split lines have exact same semantics as lines that are not split. Exceptions to this rule are [suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation), [test](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation) and [keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-documentation) documentation as well [suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata). With them split values are automatically [joined together with the newline character](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#newlines-in-test-data) to ease creating multiline values.

The ... syntax allows also splitting variables in the [Variable section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-section). When long scalar variables (e.g. ${STRING}) are split to multiple rows, the final value is got by concatenating the rows together. The separator is a space by default, but that can be changed by starting the value with SEPARATOR=<sep>.

Splitting lines is illustrated in the following two examples containing exactly same data without and with splitting.

**\*\*\* Settings \*\*\***

**Documentation** Here we have documentation for this suite.\nDocumentation is often quite long.\n\nIt can also contain multiple paragraphs.

**Default Tags** default tag 1 default tag 2 default tag 3 default tag 4 default tag 5

**\*\*\* Variable \*\*\***

${STRING} This is a long string. It has multiple sentences. It does not have newlines.

${MULTILINE} This is a long multiline string.\nThis is the second line.\nThis is the third and the last line.

@{LIST} this list is quite long and items in it can also be long

&{DICT} first=This value is pretty long. second=This value is even longer. It has two sentences.

**\*\*\* Test Cases \*\*\***

**Example**

[**Tags**] you probably do not have this many tags in real life

Do X first argument second argument third argument fourth argument fifth argument sixth argument

${var} = Get X first argument passed to this keyword is pretty long second argument passed to this keyword is long too

**\*\*\* Settings \*\*\***

**Documentation** Here we have documentation for this suite.

... Documentation is often quite long.

...

... It can also contain multiple paragraphs.

**Default Tags** default tag 1 default tag 2 default tag 3

... default tag 4 default tag 5

**\*\*\* Variable \*\*\***

${STRING} This is a long string.

... It has multiple sentences.

... It does not have newlines.

${MULTILINE} SEPARATOR=\n

... This is a long multiline string.

... This is the second line.

... This is the third and the last line.

@{LIST} this list is quite long and

... items in it can also be long

&{DICT} first=This value is pretty long.

... second=This value is even longer. It has two sentences.

**\*\*\* Test Cases \*\*\***

**Example**

[**Tags**] you probably do not have this many

... tags in real life

Do X first argument second argument third argument

... fourth argument fifth argument sixth argument

${var} = Get X

... first argument passed to this keyword is pretty long

... second argument passed to this keyword is long too

[**2.2   Creating test cases**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-206)

This section describes the overall test case syntax. Organizing test cases into [test suites](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite) using [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files) and [test suite directories](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories) is discussed in the next section.

When using Robot Framework for other automation purposes than test automation, it is recommended to create *tasks* instead of tests. The task syntax is for most parts identical to the test syntax, and the differences are explained in the [Creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks) section.

* [2.2.1   Test case syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-syntax)
  + [Basic syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-syntax)
  + [Settings in the Test Case section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#settings-in-the-test-case-section)
  + [Test case related settings in the Setting section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-related-settings-in-the-setting-section)
* [2.2.2   Using arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-arguments)
  + [Positional arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments)
  + [Default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values)
  + [Variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments)
  + [Named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments)
  + [Free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments)
  + [Named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments)
  + [Arguments embedded to keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#arguments-embedded-to-keyword-names)
* [2.2.3   Failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#failures)
  + [When test case fails](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#when-test-case-fails)
  + [Error messages](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#error-messages)
* [2.2.4   Test case name and documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation)
* [2.2.5   Tagging test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases)
  + [Reserved tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reserved-tags)
* [2.2.6   Test setup and teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown)
* [2.2.7   Test templates](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates)
  + [Basic usage](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-usage)
  + [Templates with embedded arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#templates-with-embedded-arguments)
  + [Templates with FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#templates-with-for-loops)
  + [Templates with IF/ELSE structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#templates-with-if-else-structures)
* [2.2.8   Different test case styles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-test-case-styles)
  + [Keyword-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-driven-style)
  + [Data-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style)
  + [Behavior-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#behavior-driven-style)

[**2.2.1   Test case syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-225)

[**Basic syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-226)

Test cases are constructed in test case sections from the available keywords. Keywords can be imported from [test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) or [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files), or created in the [keyword section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) of the test case file itself.

The first column in the test case section contains test case names. A test case starts from the row with something in this column and continues to the next test case name or to the end of the section. It is an error to have something between the section headers and the first test.

The second column normally has keyword names. An exception to this rule is [setting variables from keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-return-values), when the second and possibly also the subsequent columns contain variable names and a keyword name is located after them. In either case, columns after the keyword name contain possible arguments to the specified keyword.

**\*\*\* Test Cases \*\*\***

**Valid Login**

Open Login Page

Input Username demo

Input Password mode

Submit Credentials

Welcome Page Should Be Open

**Setting Variables**

Do Something first argument second argument

${value} = Get Some Value

Should Be Equal ${value} Expected value

**Note**

Although test case names can contain any character, using ? and especially \* is not generally recommended because they are considered to be [wildcards](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#wildcards) when [selecting test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases). For example, trying to run only a test with name *Example \** like --test 'Example \*' will actually run any test starting with *Example*.

[**Settings in the Test Case section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-227)

Test cases can also have their own settings. Setting names are always in the second column, where keywords normally are, and their values are in the subsequent columns. Setting names have square brackets around them to distinguish them from keywords. The available settings are listed below and explained later in this section.

*[Documentation]*

Used for specifying a [test case documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation).

*[Tags]*

Used for [tagging test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases).

*[Setup]*, *[Teardown]*

Specify [test setup and teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown).

*[Template]*

Specifies the [template keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates) to use. The test itself will contain only data to use as arguments to that keyword.

*[Timeout]*

Used for setting a [test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout). [Timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timeouts) are discussed in their own section.

**Note**

Setting names are case-insensitive, but the format used above is recommended. Settings used to be also space-insensitive, but that was deprecated in Robot Framework 3.1 and trying to use something like [T a g s] causes an error in Robot Framework 3.2. Possible spaces between brackets and the name (e.g. [ Tags ]) are still allowed.

Example test case with settings:

**\*\*\* Test Cases \*\*\***

**Test With Settings**

[**Documentation**] Another dummy test

[**Tags**] dummy owner-johndoe

Log Hello, world!

[**Test case related settings in the Setting section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-228)

The Setting section can have the following test case related settings. These settings are mainly default values for the test case specific settings listed earlier.

*Force Tags*, *Default Tags*

The forced and default values for [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases).

*Test Setup*, *Test Teardown*

The default values for [test setup and teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown).

*Test Template*

The default [template keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates) to use.

*Test Timeout*

The default value for [test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout). [Timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timeouts) are discussed in their own section.

[**2.2.2   Using arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-229)

The earlier examples have already demonstrated keywords taking different arguments, and this section discusses this important functionality more thoroughly. How to actually implement [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments) and [library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments) with different arguments is discussed in separate sections.

Keywords can accept zero or more arguments, and some arguments may have default values. What arguments a keyword accepts depends on its implementation, and typically the best place to search this information is keyword's documentation. In the examples in this section the documentation is expected to be generated using the [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) tool, but the same information is available on documentation generated by generic documentation tools such as pydoc.

[**Positional arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-230)

Most keywords have a certain number of arguments that must always be given. In the keyword documentation this is denoted by specifying the argument names separated with a comma like first, second, third. The argument names actually do not matter in this case, except that they should explain what the argument does, but it is important to have exactly the same number of arguments as specified in the documentation. Using too few or too many arguments will result in an error.

The test below uses keywords *Create Directory* and *Copy File* from the [OperatingSystem](https://robotframework.org/robotframework/latest/libraries/OperatingSystem.html) library. Their arguments are specified as path and source, destination, which means that they take one and two arguments, respectively. The last keyword, *No Operation* from [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html), takes no arguments.

**\*\*\* Test Cases \*\*\***

**Example**

Create Directory ${TEMPDIR}/stuff

Copy File ${CURDIR}/file.txt ${TEMPDIR}/stuff

No Operation

[**Default values**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-231)

Arguments often have default values which can either be given or not. In the documentation the default value is typically separated from the argument name with an equal sign like name=default value. It is possible that all the arguments have default values, but there cannot be any positional arguments after arguments with default values.

Using default values is illustrated by the example below that uses *Create File* keyword which has arguments path, content=, encoding=UTF-8. Trying to use it without any arguments or more than three arguments would not work.

**\*\*\* Test Cases \*\*\***

**Example**

Create File ${TEMPDIR}/empty.txt

Create File ${TEMPDIR}/utf-8.txt Hyvä esimerkki

Create File ${TEMPDIR}/iso-8859-1.txt Hyvä esimerkki ISO-8859-1

[**Variable number of arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-232)

It is also possible that a keyword accepts any number of arguments. These so called *varargs* can be combined with mandatory arguments and arguments with default values, but they are always given after them. In the documentation they have an asterisk before the argument name like \*varargs.

For example, *Remove Files* and *Join Paths* keywords from the [OperatingSystem](https://robotframework.org/robotframework/latest/libraries/OperatingSystem.html) library have arguments \*paths and base, \*parts, respectively. The former can be used with any number of arguments, but the latter requires at least one argument.

**\*\*\* Test Cases \*\*\***

**Example**

Remove Files ${TEMPDIR}/f1.txt ${TEMPDIR}/f2.txt ${TEMPDIR}/f3.txt

@{paths} = Join Paths ${TEMPDIR} f1.txt f2.txt f3.txt f4.txt

[**Named arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-233)

The named argument syntax makes using arguments with [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values) more flexible, and allows explicitly labeling what a certain argument value means. Technically named arguments work exactly like [keyword arguments](http://docs.python.org/tutorial/controlflow.html#keyword-arguments) in Python.

**Basic syntax**

It is possible to name an argument given to a keyword by prefixing the value with the name of the argument like arg=value. This is especially useful when multiple arguments have default values, as it is possible to name only some the arguments and let others use their defaults. For example, if a keyword accepts arguments arg1=a, arg2=b, arg3=c, and it is called with one argument arg3=override, arguments arg1 and arg2 get their default values, but arg3 gets value override. If this sounds complicated, the [named arguments example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments-example) below hopefully makes it more clear.

The named argument syntax is both case and space sensitive. The former means that if you have an argument arg, you must use it like arg=value, and neither Arg=value nor ARG=value works. The latter means that spaces are not allowed before the = sign, and possible spaces after it are considered part of the given value.

When the named argument syntax is used with [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords), the argument names must be given without the ${} decoration. For example, user keyword with arguments ${arg1}=first, ${arg2}=second must be used like arg2=override.

Using normal positional arguments after named arguments like, for example, | Keyword | arg=value | positional |, does not work. The relative order of the named arguments does not matter.

**Named arguments with variables**

It is possible to use [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) in both named argument names and values. If the value is a single [scalar variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variable), it is passed to the keyword as-is. This allows using any objects, not only strings, as values also when using the named argument syntax. For example, calling a keyword like arg=${object} will pass the variable ${object} to the keyword without converting it to a string.

If variables are used in named argument names, variables are resolved before matching them against argument names.

The named argument syntax requires the equal sign to be written literally in the keyword call. This means that variable alone can never trigger the named argument syntax, not even if it has a value like foo=bar. This is important to remember especially when wrapping keywords into other keywords. If, for example, a keyword takes a [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments) like @{args} and passes all of them to another keyword using the same @{args} syntax, possible named=arg syntax used in the calling side is not recognized. This is illustrated by the example below.

**\*\*\* Test Cases \*\*\***

**Example**

Run Program shell=True *# This will not come as a named argument to Run Process*

**\*\*\* Keywords \*\*\***

**Run Program**

[**Arguments**] @{args}

Run Process program.py @{args} *# Named arguments are not recognized from inside @{args}*

If keyword needs to accept and pass forward any named arguments, it must be changed to accept [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments). See [free named argument examples](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-argument-examples) for a wrapper keyword version that can pass both positional and named arguments forward.

**Escaping named arguments syntax**

The named argument syntax is used only when the part of the argument before the equal sign matches one of the keyword's arguments. It is possible that there is a positional argument with a literal value like foo=quux, and also an unrelated argument with name foo. In this case the argument foo either incorrectly gets the value quux or, more likely, there is a syntax error.

In these rare cases where there are accidental matches, it is possible to use the backslash character to [escape](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) the syntax like foo\=quux. Now the argument will get a literal value foo=quux. Note that escaping is not needed if there are no arguments with name foo, but because it makes the situation more explicit, it may nevertheless be a good idea.

**Where named arguments are supported**

As already explained, the named argument syntax works with keywords. In addition to that, it also works when [importing libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-libraries).

Naming arguments is supported by [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) and by most [test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries). The only exceptions are Python keywords explicitly using [positional-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-only-arguments).

**Named arguments example**

The following example demonstrates using the named arguments syntax with library keywords, user keywords, and when importing the [Telnet](https://robotframework.org/robotframework/latest/libraries/Telnet.html) test library.

**\*\*\* Settings \*\*\***

**Library** **Telnet** prompt=$ default\_log\_level=DEBUG

**\*\*\* Test Cases \*\*\***

**Example**

Open connection 10.0.0.42 port=${PORT} alias=example

List files options=-lh

List files path=/tmp options=-l

**\*\*\* Keywords \*\*\***

**List files**

[**Arguments**] ${path}=. ${options}=

Execute command ls ${options} ${path}

[**Free named arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-234)

Robot Framework supports *free named arguments*, often also called *free keyword arguments* or *kwargs*, similarly as [Python supports \*\*kwargs](http://docs.python.org/tutorial/controlflow.html#keyword-arguments). What this means is that a keyword can receive all arguments that use the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) (name=value) and do not match any arguments specified in the signature of the keyword.

Free named arguments are supported by same keyword types than [normal named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#where-named-arguments-are-supported). How keywords specify that they accept free named arguments depends on the keyword type. For example, [Python based keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-keyword-arguments-kwargs) simply use \*\*kwargs and [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-user-keywords) use &{kwargs}.

Free named arguments support variables similarly as [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments-with-variables). In practice that means that variables can be used both in names and values, but the escape sign must always be visible literally. For example, both foo=${bar} and ${foo}=${bar} are valid, as long as the variables that are used exist. An extra limitation is that free argument names must always be strings.

**Examples**

As the first example of using free named arguments, let's take a look at *Run Process* keyword in the [Process](https://robotframework.org/robotframework/latest/libraries/Process.html) library. It has a signature command, \*arguments, \*\*configuration, which means that it takes the command to execute (command), its arguments as [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments) (\*arguments) and finally optional configuration parameters as free named arguments (\*\*configuration). The example below also shows that variables work with free keyword arguments exactly like when [using the named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments-with-variables).

**\*\*\* Test Cases \*\*\***

**Free Named Arguments**

Run Process program.py arg1 arg2 cwd=/home/user

Run Process program.py argument shell=True env=${ENVIRON}

See [Free keyword arguments (\*\*kwargs)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-keyword-arguments-kwargs) section under [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) for more information about using the free named arguments syntax in your custom test libraries.

As the second example, let's create a wrapper [user keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) for running the program.py in the above example. The wrapper keyword *Run Program* accepts all positional and named arguments and passes them forward to *Run Process* along with the name of the command to execute.

**\*\*\* Test Cases \*\*\***

**Free Named Arguments**

Run Program arg1 arg2 cwd=/home/user

Run Program argument shell=True env=${ENVIRON}

**\*\*\* Keywords \*\*\***

**Run Program**

[**Arguments**] @{args} &{config}

Run Process program.py @{args} &{config}

[**Named-only arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-235)

Starting from Robot Framework 3.1, keywords can accept argument that must always be named using the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax). If, for example, a keyword would accept a single named-only argument example, it would always need to be used like example=value and using just value would not work. This syntax is inspired by the [keyword-only arguments](https://www.python.org/dev/peps/pep-3102) syntax supported by Python 3.

For most parts named-only arguments work the same way as [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments). The main difference is that libraries implemented with Python 2 using the [static library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-keywords) [do not support this syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-only-arguments).

As an example of using the [named-only arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-user-keywords), here is a variation of the *Run Program* in the above [free named argument examples](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-argument-examples) that only supports configuring shell:

**\*\*\* Test Cases \*\*\***

**Named-only Arguments**

Run Program arg1 arg2 *# 'shell' is False (default)*

Run Program argument shell=True *# 'shell' is True*

**\*\*\* Keywords \*\*\***

**Run Program**

[**Arguments**] @{args} ${shell}=False

Run Process program.py @{args} shell=${shell}

[**Arguments embedded to keyword names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-236)

A totally different approach to specify arguments is embedding them into keyword names. This syntax is supported by both [test library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-names) and [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-name).

[**2.2.3   Failures**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-237)

[**When test case fails**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-238)

A test case fails if any of the keyword it uses fails. Normally this means that execution of that test case is stopped, possible [test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) is executed, and then execution continues from the next test case. It is also possible to use special [continuable failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure) if stopping test execution is not desired.

[**Error messages**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-239)

The error message assigned to a failed test case is got directly from the failed keyword. Often the error message is created by the keyword itself, but some keywords allow configuring them.

In some circumstances, for example when continuable failures are used, a test case can fail multiple times. In that case the final error message is got by combining the individual errors. Very long error messages are [automatically cut from the middle](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-error-message-length-in-reports) to keep [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) easier to read, but full error messages are always visible in [log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) as messages of the failed keywords.

By default error messages are normal text, but they can [contain HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#html-in-error-messages). This is enabled by starting the error message with marker string \*HTML\*. This marker will be removed from the final error message shown in reports and logs. Using HTML in a custom message is shown in the second example below.

**\*\*\* Test Cases \*\*\***

**Normal Error**

Fail This is a rather boring example...

**HTML Error**

${number} = Get Number

Should Be Equal ${number} 42 \*HTML\* Number is not my <b>MAGIC</b> number.

[**2.2.4   Test case name and documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-240)

The test case name comes directly from the Test Case section: it is exactly what is entered into the test case column. Test cases in one test suite should have unique names. Pertaining to this, you can also use the [automatic variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatic-variables) ${TEST\_NAME} within the test itself to refer to the test name. It is available whenever a test is being executed, including all user keywords, as well as the test setup and the test teardown.

Starting from Robot Framework 3.2, possible [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) in the test case name are resolved so that the final name will contain the variable value. If the variable does not exist, its name is left unchanged.

**\*\*\* Variables \*\*\***

${MAX AMOUNT} ${5000000}

**\*\*\* Test Cases \*\*\***

**Amount cannot be larger than** ${MAX AMOUNT}

*# ...*

The *[Documentation]* setting allows you to set a free documentation for a test case. That text is shown in the command line output, as well as the resulting test logs and test reports. It is possible to use simple [HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) in documentation and [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) can be used to make the documentation dynamic. Possible non-existing variables are left unchanged.

If documentation is split into multiple columns, cells in one row are concatenated together with spaces. If documentation is [split into multiple rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows), the created documentation lines themselves are [concatenated using newlines](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#newlines-in-test-data). Newlines are not added if a line already ends with a newline or an [escaping backslash](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping).

**\*\*\* Test Cases \*\*\***

**Simple**

[**Documentation**] Simple documentation

No Operation

**Formatting**

[**Documentation**] \*This is bold\*, \_this is italic\_ and here is a link: http://robotframework.org

No Operation

**Variables**

[**Documentation**] Executed at ${HOST} by ${USER}

No Operation

**Splitting**

[**Documentation**] This documentation is split into multiple columns

No Operation

**Many lines**

[**Documentation**] Here we have

... an automatic newline

No Operation

It is important that test cases have clear and descriptive names, and in that case they normally do not need any documentation. If the logic of the test case needs documenting, it is often a sign that keywords in the test case need better names and they are to be enhanced, instead of adding extra documentation. Finally, metadata, such as the environment and user information in the last example above, is often better specified using [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases).

[**2.2.5   Tagging test cases**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-241)

Using tags in Robot Framework is a simple, yet powerful mechanism for classifying test cases. Tags are free text and they can be used at least for the following purposes:

* Tags are shown in test [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file), [logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) and, of course, in the test data, so they provide metadata to test cases.
* [Statistics](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-statistics) about test cases (total, passed, failed are automatically collected based on tags).
* With tags, you can [include or exclude](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) test cases to be executed.
* With tags, you can specify which test cases should be [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped).

In this section it is only explained how to set tags for test cases, and different ways to do it are listed below. These approaches can naturally be used together.

*Force Tags* in the Setting section

All test cases in a test case file with this setting always get specified tags. If it is used in the test suite initialization file, all test cases in sub test suites get these tags.

*Default Tags* in the Setting section

Test cases that do not have a *[Tags]* setting of their own get these tags. Default tags are not supported in test suite initialization files.

*[Tags]* in the Test Case section

A test case always gets these tags. Additionally, it does not get the possible tags specified with *Default Tags*, so it is possible to override the *Default Tags* by using empty value. It is also possible to use value NONE to override default tags.

--settag command line option

All executed test cases get tags set with this option in addition to tags they got elsewhere.

*Set Tags*, *Remove Tags*, *Fail* and *Pass Execution* keywords

These [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords can be used to manipulate tags dynamically during the test execution.

Tags are free text, but they are normalized so that they are converted to lowercase and all spaces are removed. If a test case gets the same tag several times, other occurrences than the first one are removed. Tags can be created using variables, assuming that those variables exist.

**\*\*\* Settings \*\*\***

**Force Tags** req-42

**Default Tags** owner-john smoke

**\*\*\* Variables \*\*\***

${HOST} 10.0.1.42

**\*\*\* Test Cases \*\*\***

**No own tags**

[**Documentation**] This test has tags owner-john, smoke and req-42.

No Operation

**With own tags**

[**Documentation**] This test has tags not\_ready, owner-mrx and req-42.

[**Tags**] owner-mrx not\_ready

No Operation

**Own tags with variables**

[**Documentation**] This test has tags host-10.0.1.42 and req-42.

[**Tags**] host-${HOST}

No Operation

**Empty own tags**

[**Documentation**] This test has only tag req-42.

[**Tags**]

No Operation

**Set Tags and Remove Tags Keywords**

[**Documentation**] This test has tags mytag and owner-john.

Set Tags mytag

Remove Tags smoke req-\*

[**Reserved tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-242)

Users are generally free to use whatever tags that work in their context. There are, however, certain tags that have a predefined meaning for Robot Framework itself, and using them for other purposes can have unexpected results. All special tags Robot Framework has and will have in the future have the robot: prefix. To avoid problems, users should thus not use any tag with this prefixes unless actually activating the special functionality.

At the time of writing, the only special tags are robot:exit, that is automatically added to tests when [stopping test execution gracefully](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution-gracefully), and robot:no-dry-run, that can be used to disable the [dry run](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dry-run) mode as well as robot:continue-on-failure which controls continuable execution. More usages are likely to be added in the future.

As of RobotFramework 4.1, reserved tags are suppressed by default in the test suite's tag statistics. They will be shown when they are explicitly included via the --tagstatinclude 'robot:\*' command line option.

As of RobotFramework 5.0, new reserved tags include robot:skip, robot:skip-on-failure and robot:exclude.

[**2.2.6   Test setup and teardown**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-243)

Robot Framework has similar test setup and teardown functionality as many other test automation frameworks. In short, a test setup is something that is executed before a test case, and a test teardown is executed after a test case. In Robot Framework setups and teardowns are just normal keywords with possible arguments.

Setup and teardown are always a single keyword. If they need to take care of multiple separate tasks, it is possible to create higher-level [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) for that purpose. An alternative solution is executing multiple keywords using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keywords*.

The test teardown is special in two ways. First of all, it is executed also when a test case fails, so it can be used for clean-up activities that must be done regardless of the test case status. In addition, all the keywords in the teardown are also executed even if one of them fails. This [continue on failure](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure) functionality can be used also with normal keywords, but inside teardowns it is on by default.

The easiest way to specify a setup or a teardown for test cases in a test case file is using the *Test Setup* and *Test Teardown* settings in the Setting section. Individual test cases can also have their own setup or teardown. They are defined with the *[Setup]* or *[Teardown]* settings in the test case section and they override possible *Test Setup* and *Test Teardown* settings. Having no keyword after a *[Setup]* or *[Teardown]* setting means having no setup or teardown. It is also possible to use value NONE to indicate that a test has no setup/teardown.

**\*\*\* Settings \*\*\***

**Test Setup** Open Application App A

**Test Teardown** Close Application

**\*\*\* Test Cases \*\*\***

**Default values**

[**Documentation**] Setup and teardown from setting section

Do Something

**Overridden setup**

[**Documentation**] Own setup, teardown from setting section

[**Setup**] Open Application App B

Do Something

**No teardown**

[**Documentation**] Default setup, no teardown at all

Do Something

[**Teardown**]

**No teardown 2**

[**Documentation**] Setup and teardown can be disabled also with special value NONE

Do Something

[**Teardown**] NONE

**Using variables**

[**Documentation**] Setup and teardown specified using variables

[**Setup**] ${SETUP}

Do Something

[**Teardown**] ${TEARDOWN}

The name of the keyword to be executed as a setup or a teardown can be a variable. This facilitates having different setups or teardowns in different environments by giving the keyword name as a variable from the command line.

**Note**

[Test suites can have a setup and teardown of their own](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown). A suite setup is executed before any test cases or sub test suites in that test suite, and similarly a suite teardown is executed after them.

[**2.2.7   Test templates**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-244)

Test templates convert normal [keyword-driven](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-driven-style) test cases into [data-driven](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style) tests. Whereas the body of a keyword-driven test case is constructed from keywords and their possible arguments, test cases with template contain only the arguments for the template keyword. Instead of repeating the same keyword multiple times per test and/or with all tests in a file, it is possible to use it only per test or just once per file.

Template keywords can accept both normal positional and named arguments, as well as arguments embedded to the keyword name. Unlike with other settings, it is not possible to define a template using a variable.

[**Basic usage**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-245)

How a keyword accepting normal positional arguments can be used as a template is illustrated by the following example test cases. These two tests are functionally fully identical.

**\*\*\* Test Cases \*\***

**Normal test case**

Example keyword first argument second argument

**Templated test case**

[**Template**] Example keyword

first argument second argument

As the example illustrates, it is possible to specify the template for an individual test case using the *[Template]* setting. An alternative approach is using the *Test Template* setting in the Setting section, in which case the template is applied for all test cases in that test case file. The *[Template]* setting overrides the possible template set in the Setting section, and an empty value for *[Template]* means that the test has no template even when *Test Template* is used. It is also possible to use value NONE to indicate that a test has no template.

If a templated test case has multiple data rows in its body, the template is applied for all the rows one by one. This means that the same keyword is executed multiple times, once with data on each row. Templated tests are also special so that all the rounds are executed even if one or more of them fails. It is possible to use this kind of [continue on failure](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure) mode with normal tests too, but with the templated tests the mode is on automatically.

**\*\*\* Settings \*\*\***

**Test Template** Example keyword

**\*\*\* Test Cases \*\*\***

**Templated test case**

first round 1 first round 2

second round 1 second round 2

third round 1 third round 2

Using keywords with [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values) or accepting [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments), as well as using [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments) and [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments), work with templates exactly like they work otherwise. Using [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) in arguments is also supported normally.

[**Templates with embedded arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-246)

Templates support a variation of the [embedded argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedded-argument-syntax). With templates this syntax works so that if the template keyword has variables in its name, they are considered placeholders for arguments and replaced with the actual arguments used with the template. The resulting keyword is then used without positional arguments. This is best illustrated with an example:

**\*\*\* Test Cases \*\*\***

**Normal test case with embedded arguments**

The result of 1 + 1 should be 2

The result of 1 + 2 should be 3

**Template with embedded arguments**

[**Template**] The result of ${calculation} should be ${expected}

1 + 1 2

1 + 2 3

**\*\*\* Keywords \*\*\***

**The result of** ${calculation} **should be** ${expected}

${result} = Calculate ${calculation}

Should Be Equal ${result} ${expected}

When embedded arguments are used with templates, the number of arguments in the template keyword name must match the number of arguments it is used with. The argument names do not need to match the arguments of the original keyword, though, and it is also possible to use different arguments altogether:

**\*\*\* Test Cases \*\*\***

**Different argument names**

[**Template**] The result of ${foo} should be ${bar}

1 + 1 2

1 + 2 3

**Only some arguments**

[**Template**] The result of ${calculation} should be 3

1 + 2

4 - 1

**New arguments**

[**Template**] The ${meaning} of ${life} should be 42

result 21 \* 2

The main benefit of using embedded arguments with templates is that argument names are specified explicitly. When using normal arguments, the same effect can be achieved by naming the columns that contain arguments. This is illustrated by the [data-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style) example in the next section.

[**Templates with FOR loops**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-247)

If templates are used with [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops), the template is applied for all the steps inside the loop. The continue on failure mode is in use also in this case, which means that all the steps are executed with all the looped elements even if there are failures.

**\*\*\* Test Cases \*\*\***

**Template with FOR loop**

[**Template**] Example keyword

FOR ${item} IN @{ITEMS}

${item} 2nd arg

END

FOR ${index} IN RANGE 42

1st arg ${index}

END

[**Templates with IF/ELSE structures**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-248)

[IF/ELSE structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else-structures) can be also used together with templates. This can be useful, for example, when used together with [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) to filter executed arguments.

**\*\*\* Test Cases \*\*\***

**Template with FOR and IF**

[**Template**] Example keyword

FOR ${item} IN @{ITEMS}

IF ${item} < 5

${item} 2nd arg

END

END

[**2.2.8   Different test case styles**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-249)

There are several different ways in which test cases may be written. Test cases that describe some kind of *workflow* may be written either in keyword-driven or behavior-driven style. Data-driven style can be used to test the same workflow with varying input data.

[**Keyword-driven style**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-250)

Workflow tests, such as the *Valid Login* test described [earlier](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#example-tests), are constructed from several keywords and their possible arguments. Their normal structure is that first the system is taken into the initial state (*Open Login Page* in the *Valid Login* example), then something is done to the system (*Input Name*, *Input Password*, *Submit Credentials*), and finally it is verified that the system behaved as expected (*Welcome Page Should Be Open*).

[**Data-driven style**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-251)

Another style to write test cases is the *data-driven* approach where test cases use only one higher-level keyword, often created as a [user keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords), that hides the actual test workflow. These tests are very useful when there is a need to test the same scenario with different input and/or output data. It would be possible to repeat the same keyword with every test, but the [test template](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates) functionality allows specifying the keyword to use only once.

**\*\*\* Settings \*\*\***

**Test Template** Login with invalid credentials should fail

**\*\*\* Test Cases \*\*\*** **USERNAME** **PASSWORD**

**Invalid User Name** invalid ${VALID PASSWORD}

**Invalid Password** ${VALID USER} invalid

**Invalid User Name and Password** invalid invalid

**Empty User Name** ${EMPTY} ${VALID PASSWORD}

**Empty Password** ${VALID USER} ${EMPTY}

**Empty User Name and Password** ${EMPTY} ${EMPTY}

**Tip**

Naming columns like in the example above makes tests easier to understand. This is possible because on the header row other cells except the first one [are ignored](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-sections).

The above example has six separate tests, one for each invalid user/password combination, and the example below illustrates how to have only one test with all the combinations. When using [test templates](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates), all the rounds in a test are executed even if there are failures, so there is no real functional difference between these two styles. In the above example separate combinations are named so it is easier to see what they test, but having potentially large number of these tests may mess-up statistics. Which style to use depends on the context and personal preferences.

**\*\*\* Test Cases \*\*\***

**Invalid Password**

[**Template**] Login with invalid credentials should fail

invalid ${VALID PASSWORD}

${VALID USER} invalid

invalid whatever

${EMPTY} ${VALID PASSWORD}

${VALID USER} ${EMPTY}

${EMPTY} ${EMPTY}

[**Behavior-driven style**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-252)

It is also possible to write test cases as requirements that also non-technical project stakeholders must understand. These *executable requirements* are a corner stone of a process commonly called [Acceptance Test Driven Development](http://testobsessed.com/2008/12/08/acceptance-test-driven-development-atdd-an-overview) (ATDD) or [Specification by Example](http://en.wikipedia.org/wiki/Specification_by_example).

One way to write these requirements/tests is *Given-When-Then* style popularized by [Behavior Driven Development](http://en.wikipedia.org/wiki/Behavior_Driven_Development) (BDD). When writing test cases in this style, the initial state is usually expressed with a keyword starting with word *Given*, the actions are described with keyword starting with *When* and the expectations with a keyword starting with *Then*. Keyword starting with *And* or *But* may be used if a step has more than one action.

**\*\*\* Test Cases \*\*\***

**Valid Login**

*Given* login page is open

*When* valid username and password are inserted

*and* credentials are submitted

*Then* welcome page should be open

**Ignoring *Given/When/Then/And/But* prefixes**

Prefixes *Given*, *When*, *Then*, *And* and *But* are dropped when matching keywords are searched, if no match with the full name is found. This works for both user keywords and library keywords. For example, *Given login page is open* in the above example can be implemented as user keyword either with or without the word *Given*. Ignoring prefixes also allows using the same keyword with different prefixes. For example *Welcome page should be open* could also used as *And welcome page should be open*.

**Embedding data to keywords**

When writing concrete examples it is useful to be able to pass actual data to keyword implementations. User keywords support this by allowing [embedding arguments into keyword name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-name).

[**2.3   Creating tasks**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-207)

In addition to test automation, Robot Framework can be used for other automation purposes, including [robotic process automation](https://en.wikipedia.org/wiki/Robotic_process_automation) (RPA). It has always been possible, but Robot Framework 3.1 added official support for automating *tasks*, not only tests. For most parts creating tasks works the same way as [creating tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tests) and the only real difference is in terminology. Tasks can also be organized into [suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-suites) exactly like test cases.

* [2.3.1   Task syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-syntax)
* [2.3.2   Task related settings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-related-settings)

[**2.3.1   Task syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-253)

Tasks are created based on the available keywords exactly like test cases, and the task syntax is in general identical to the [test case syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-syntax). The main difference is that tasks are created in Task sections instead of Test Case sections:

**\*\*\* Tasks \*\*\***

**Process invoice**

Read information from PDF

Validate information

Submit information to backend system

Validate information is visible in web UI

It is an error to have both tests and tasks in same file.

[**2.3.2   Task related settings**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-254)

Settings that can be used in the task section are exactly the same as in the [test case section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#settings-in-the-test-case-section). In the [setting section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-related-settings-in-the-setting-section) it is possible to use *Task Setup*, *Task Teardown*, *Task Template* and *Task Timeout* instead of their *Test* variants.

[**2.4   Creating test suites**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-208)

Robot Framework [test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-cases) are created in test case files, which can be organized into directories. These files and directories create a hierarchical test suite structure. Same concepts apply also when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks), but the terminology differs.

* [2.4.1   Test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files)
* [2.4.2   Test suite directories](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories)
  + [Initialization files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files)
* [2.4.3   Test suite name and documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation)
* [2.4.4   Free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata)
* [2.4.5   Suite setup and teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown)

[**2.4.1   Test case files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-255)

Robot Framework test cases [are created](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-cases) using test case sections in test case files. Such a file automatically creates a test suite from all the test cases it contains. There is no upper limit for how many test cases there can be, but it is recommended to have less than ten, unless the [data-driven approach](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style) is used, where one test case consists of only one high-level keyword.

The following settings in the Setting section can be used to customize the test suite:

*Documentation*

Used for specifying a [test suite documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation)

*Metadata*

Used for setting [free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) as name-value pairs.

*Suite Setup*, *Suite Teardown*

Specify [suite setup and teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown).

**Note**

Setting names are case-insensitive, but the format used above is recommended.

[**2.4.2   Test suite directories**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-256)

Test case files can be organized into directories, and these directories create higher-level test suites. A test suite created from a directory cannot have any test cases directly, but it contains other test suites with test cases, instead. These directories can then be placed into other directories creating an even higher-level suite. There are no limits for the structure, so test cases can be organized as needed.

When a test directory is executed, the files and directories it contains are processed recursively as follows:

* Files and directories with names starting with a dot (*.*) or an underscore (*\_*) are ignored.
* Directories with the name *CVS* are ignored (case-sensitive).
* Files in [supported file formats](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats) are processed.
* Other files are ignored.

If a file or directory that is processed does not contain any test cases, it is silently ignored (a message is written to the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log)) and the processing continues.

[**Initialization files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-257)

A test suite created from a directory can have similar settings as a suite created from a test case file. Because a directory alone cannot have that kind of information, it must be placed into a special test suite initialization file. An initialization file name must always be of the format *\_\_init\_\_.ext*, where the extension must be one of the [supported file formats](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats) (typically *\_\_init\_\_.robot*). The name format is borrowed from Python, where files named in this manner denote that a directory is a module.

Initialization files have the same structure and syntax as test case files, except that they cannot have test case sections and not all settings are supported. Variables and keywords created or imported in initialization files *are not* available in the lower level test suites. If you need to share variables or keywords, you can put them into [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) that can be imported both by initialization and test case files.

The main usage for initialization files is specifying test suite related settings similarly as in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files), but setting some [test case related settings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-related-settings-in-the-setting-section) is also possible. How to use different settings in the initialization files is explained below.

*Documentation*, *Metadata*, *Suite Setup*, *Suite Teardown*

These test suite specific settings work the same way as in test case files.

*Force Tags*

Specified tags are unconditionally set to all test cases in all test case files this directory contains directly or recursively.

*Test Setup*, *Test Teardown*, *Test Timeout*

Set the default value for test setup/teardown or test timeout to all test cases this directory contains. Can be overridden on lower level. Notice that keywords used as setups and teardowns must be available in test case files where tests using them are. Defining keywords in the initialization file itself is not enough.

*Task Setup*, *Task Teardown*, *Task Timeout*

Aliases for *Test Setup*, *Test Teardown*, and *Test Timeout*, respectively, that can be used when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks), not tests.

*Default Tags*, *Test Template*

Not supported in initialization files.

**\*\*\* Settings \*\*\***

**Documentation** Example suite

**Suite Setup** Do Something ${MESSAGE}

**Force Tags** example

**Library** **SomeLibrary**

**\*\*\* Variables \*\*\***

${MESSAGE} Hello, world!

**\*\*\* Keywords \*\*\***

**Do Something**

[**Arguments**] ${args}

Some Keyword ${arg}

Another Keyword

[**2.4.3   Test suite name and documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-258)

The test suite name is constructed from the file or directory name. The name is created so that the extension is ignored, possible underscores are replaced with spaces, and names fully in lower case are title cased. For example, *some\_tests.robot* becomes *Some Tests* and *My\_test\_directory* becomes *My test directory*.

The file or directory name can contain a prefix to control the [execution order](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#execution-order) of the suites. The prefix is separated from the base name by two underscores and, when constructing the actual test suite name, both the prefix and underscores are removed. For example files *01\_\_some\_tests.robot* and *02\_\_more\_tests.robot* create test suites *Some Tests* and *More Tests*, respectively, and the former is executed before the latter.

The documentation for a test suite is set using the *Documentation* setting in the Setting section. It can be used in test case files or, with higher-level suites, in test suite initialization files. Test suite documentation has exactly the same characteristics regarding to where it is shown and how it can be created as [test case documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation).

**\*\*\* Settings \*\*\***

**Documentation** An example test suite documentation with \*some\* \_formatting\_.

... See test documentation for more documentation examples.

Both the name and documentation of the top-level test suite can be overridden in test execution. This can be done with the command line options --name and --doc, respectively, as explained in section [Setting metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-metadata).

[**2.4.4   Free test suite metadata**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-259)

Test suites can also have other metadata than the documentation. This metadata is defined in the Setting section using the *Metadata* setting. Metadata set in this manner is shown in test reports and logs.

The name and value for the metadata are located in the columns following *Metadata*. The value is handled similarly as documentation, which means that it can be split [into several cells](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows) (joined together with spaces) or [into several rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#newlines-in-test-data) (joined together with newlines), simple [HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) works and even [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) can be used.

**\*\*\* Settings \*\*\***

**Metadata** Version 2.0

**Metadata** More Info For more information about \*Robot Framework\* see http://robotframework.org

**Metadata** Executed At ${HOST}

For top-level test suites, it is possible to set metadata also with the --metadata command line option. This is discussed in more detail in section [Setting metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-metadata).

[**2.4.5   Suite setup and teardown**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-260)

Not only [test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) but also test suites can have a setup and a teardown. A suite setup is executed before running any of the suite's test cases or child test suites, and a test teardown is executed after them. All test suites can have a setup and a teardown; with suites created from a directory they must be specified in a [test suite initialization file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files).

Similarly as with test cases, a suite setup and teardown are keywords that may take arguments. They are defined in the Setting section with *Suite Setup* and *Suite Teardown* settings, respectively. Keyword names and possible arguments are located in the columns after the setting name.

If a suite setup fails, all test cases in it and its child test suites are immediately assigned a fail status and they are not actually executed. This makes suite setups ideal for checking preconditions that must be met before running test cases is possible.

A suite teardown is normally used for cleaning up after all the test cases have been executed. It is executed even if the setup of the same suite fails. If the suite teardown fails, all test cases in the suite are marked failed, regardless of their original execution status. Note that all the keywords in suite teardowns are executed even if one of them fails.

The name of the keyword to be executed as a setup or a teardown can be a variable. This facilitates having different setups or teardowns in different environments by giving the keyword name as a variable from the command line.

[**2.5   Using test libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-209)

Test libraries contain those lowest-level keywords, often called *library keywords*, which actually interact with the system under test. All test cases always use keywords from some library, often through higher-level [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords). This section explains how to take test libraries into use and how to use the keywords they provide. [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) is described in a separate section.

* [2.5.1   Importing libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-libraries)
  + [Using Library setting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-library-setting)
  + [Using Import Library keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-import-library-keyword)
* [2.5.2   Specifying library to import](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-library-to-import)
  + [Using library name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-library-name)
  + [Using physical path to library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-physical-path-to-library)
* [2.5.3   Setting custom name to test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name-to-test-library)
* [2.5.4   Standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#standard-libraries)
  + [Normal standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#normal-standard-libraries)
  + [Remote library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library)
* [2.5.5   External libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#external-libraries)

[**2.5.1   Importing libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-261)

Test libraries are typically imported using the *Library* setting, but it is also possible to use the *Import Library* keyword.

[**Using Library setting**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-262)

Test libraries are normally imported using the *Library* setting in the Setting section and having the library name in the subsequent column. Unlike most of the other data, the library name is both case- and space-sensitive. If a library is in a package, the full name including the package name must be used.

In those cases where the library needs arguments, they are listed in the columns after the library name. It is possible to use default values, variable number of arguments, and named arguments in test library imports similarly as with [arguments to keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-arguments). Both the library name and arguments can be set using variables.

**\*\*\* Settings \*\*\***

**Library** **OperatingSystem**

**Library** **my.package.TestLibrary**

**Library** **MyLibrary** arg1 arg2

**Library** ${LIBRARY}

It is possible to import test libraries in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files), [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) and [test suite initialization files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files). In all these cases, all the keywords in the imported library are available in that file. With resource files, those keywords are also available in other files using them.

[**Using Import Library keyword**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-263)

Another possibility to take a test library into use is using the keyword *Import Library* from the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library. This keyword takes the library name and possible arguments similarly as the *Library* setting. Keywords from the imported library are available in the test suite where the *Import Library* keyword was used. This approach is useful in cases where the library is not available when the test execution starts and only some other keywords make it available.

**\*\*\* Test Cases \*\*\***

**Example**

Do Something

Import Library MyLibrary arg1 arg2

KW From MyLibrary

[**2.5.2   Specifying library to import**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-264)

Libraries to import can be specified either by using the library name or the path to the library. These approaches work the same way regardless if the library is imported using the *Library* setting or the *Import Library* keyword.

[**Using library name**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-265)

The most common way to specify a test library to import is using its name, like it has been done in all the examples in this section. In these cases Robot Framework tries to find the class or module implementing the library from the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path). Libraries that are installed somehow ought to be in the module search path automatically, but with other libraries the search path may need to be configured separately.

The biggest benefit of this approach is that when the module search path has been configured, often using a custom [start-up script](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#start-up-script), normal users do not need to think where libraries actually are installed. The drawback is that getting your own, possible very simple, libraries into the search path may require some additional configuration.

[**Using physical path to library**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-266)

Another mechanism for specifying the library to import is using a path to it in the file system. This path is considered relative to the directory where current test data file is situated similarly as paths to [resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files). The main benefit of this approach is that there is no need to configure the module search path.

If the library is a file, the path to it must contain extension, i.e. *.py*. If a library is implemented as a directory, the path to it must have a trailing forward slash (/) if the path is relative. With absolute paths the trailing slash is optional. Following examples demonstrate these different usages.

**\*\*\* Settings \*\*\***

**Library** **PythonLibrary.py**

**Library** **relative/path/PythonDirLib/** possible arguments

**Library** ${RESOURCES}**/Example.class**

A limitation of this approach is that libraries implemented as Python classes [must be in a module with the same name as the class](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-name).

[**2.5.3   Setting custom name to test library**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-267)

The library name is shown in test logs before keyword names, and if multiple keywords have the same name, they must be used so that the [keyword name is prefixed with the library name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-keywords-with-same-names). The library name is got normally from the module or class name implementing it, but there are some situations where changing it is desirable:

* There is a need to import the same library several times with different arguments. This is not possible otherwise.
* The library name is inconveniently long.
* You want to use variables to import different libraries in different environments, but refer to them with the same name.
* The library name is misleading or otherwise poor. In this case, changing the actual name is, of course, a better solution.

The basic syntax for specifying the new name is having the text WITH NAME (case-sensitive) after the library name and then having the new name in the next cell. The specified name is shown in logs and must be used in the test data when using keywords' full name (*LibraryName.Keyword Name*).

**\*\*\* Settings \*\*\***

**Library** **com.company.TestLib** WITH NAME TestLib

**Library** ${LIBRARY} WITH NAME MyName

Possible arguments to the library are placed into cells between the original library name and the WITH NAME text. The following example illustrates how the same library can be imported several times with different arguments:

**\*\*\* Settings \*\*\***

**Library** **SomeLibrary** localhost 1234 WITH NAME LocalLib

**Library** **SomeLibrary** server.domain 8080 WITH NAME RemoteLib

**\*\*\* Test Cases \*\*\***

**My Test**

LocalLib.Some Keyword some arg second arg

RemoteLib.Some Keyword another arg whatever

LocalLib.Another Keyword

Setting a custom name to a test library works both when importing a library in the Setting section and when using the *Import Library* keyword.

[**2.5.4   Standard libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-268)

Some test libraries are distributed with Robot Framework and these libraries are called *standard libraries*. The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library is special, because it is taken into use automatically and thus its keywords are always available. Other standard libraries need to be imported in the same way as any other libraries, but there is no need to install them.

[**Normal standard libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-269)

The available normal standard libraries are listed below with links to their documentations:

* [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html)
* [Collections](https://robotframework.org/robotframework/latest/libraries/Collections.html)
* [DateTime](https://robotframework.org/robotframework/latest/libraries/DateTime.html)
* [Dialogs](https://robotframework.org/robotframework/latest/libraries/Dialogs.html)
* [OperatingSystem](https://robotframework.org/robotframework/latest/libraries/OperatingSystem.html)
* [Process](https://robotframework.org/robotframework/latest/libraries/Process.html)
* [Screenshot](https://robotframework.org/robotframework/latest/libraries/Screenshot.html)
* [String](https://robotframework.org/robotframework/latest/libraries/String.html)
* [Telnet](https://robotframework.org/robotframework/latest/libraries/Telnet.html)
* [XML](https://robotframework.org/robotframework/latest/libraries/XML.html)

[**Remote library**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-270)

In addition to the normal standard libraries listed above, there is also *Remote* library that is totally different than the other standard libraries. It does not have any keywords of its own but it works as a proxy between Robot Framework and actual test library implementations. These libraries can be running on other machines than the core framework and can even be implemented using languages not supported by Robot Framework natively.

See separate [Remote library interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library-interface) section for more information about this concept.

[**2.5.5   External libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-271)

Any test library that is not one of the standard libraries is, by definition, *an external library*. The Robot Framework open source community has implemented several generic libraries, such as [SeleniumLibrary](https://github.com/robotframework/SeleniumLibrary) and [SwingLibrary](https://github.com/robotframework/SwingLibrary), which are not packaged with the core framework. A list of publicly available libraries can be found from [http://robotframework.org](http://robotframework.org/).

Generic and custom libraries can obviously also be implemented by teams using Robot Framework. See [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) section for more information about that topic.

Different external libraries can have a totally different mechanism for installing them and taking them into use. Sometimes they may also require some other dependencies to be installed separately. All libraries should have clear installation and usage documentation and they should preferably automate the installation process.

[**2.6   Variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-210)

* [2.6.1   Introduction](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#introduction-1)
* [2.6.2   Using variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-variables)
  + [Scalar variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variable-syntax)
  + [List variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable-syntax)
  + [Dictionary variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable-syntax)
  + [Accessing list and dictionary items](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#accessing-list-and-dictionary-items)
  + [Environment variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#environment-variables)
* [2.6.3   Creating variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-variables)
  + [Variable section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-section)
  + [Variable file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-file)
  + [Setting variables in command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line)
  + [Return values from keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords)
  + [Using *Set Test/Suite/Global Variable* keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-set-test-suite-global-variable-keywords)
* [2.6.4   Built-in variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables)
  + [Operating-system variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#operating-system-variables)
  + [Number variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#number-variables)
  + [Boolean and None/null variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#boolean-and-none-null-variables)
  + [Space and empty variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#space-and-empty-variables)
  + [Automatic variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatic-variables)
* [2.6.5   Variable priorities and scopes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-priorities-and-scopes)
  + [Variable priorities](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-priorities)
  + [Variable scopes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-scopes)
* [2.6.6   Advanced variable features](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#advanced-variable-features)
  + [Extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax)
  + [Extended variable assignment](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-assignment)
  + [Variables inside variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables-inside-variables)
  + [Inline Python evaluation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-python-evaluation-1)

[**2.6.1   Introduction**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-272)

Variables are an integral feature of Robot Framework, and they can be used in most places in test data. Most commonly, they are used in arguments for keywords in Test Case and Keyword sections, but also all settings allow variables in their values. A normal keyword name *cannot* be specified with a variable, but the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword* can be used to get the same effect.

Robot Framework has its own variables that can be used as [scalars](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variables), [lists](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables) or [dictionaries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variables) using syntax ${SCALAR}, @{LIST} and &{DICT}, respectively. In addition to this, [environment variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#environment-variables) can be used directly with syntax %{ENV\_VAR}.

Variables are useful, for example, in these cases:

* When strings change often in the test data. With variables you only need to make these changes in one place.
* When creating system-independent and operating-system-independent test data. Using variables instead of hard-coded strings eases that considerably (for example, ${RESOURCES} instead of c:\resources, or ${HOST} instead of 10.0.0.1:8080). Because variables can be [set from the command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) when tests are started, changing system-specific variables is easy (for example, --variable HOST:10.0.0.2:1234 --variable RESOURCES:/opt/resources). This also facilitates localization testing, which often involves running the same tests with different strings.
* When there is a need to have objects other than strings as arguments for keywords. This is not possible without variables.
* When different keywords, even in different test libraries, need to communicate. You can assign a return value from one keyword to a variable and pass it as an argument to another.
* When values in the test data are long or otherwise complicated. For example, ${URL} is shorter than http://long.domain.name:8080/path/to/service?foo=1&bar=2&zap=42.

If a non-existent variable is used in the test data, the keyword using it fails. If the same syntax that is used for variables is needed as a literal string, it must be [escaped with a backslash](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) as in \${NAME}.

[**2.6.2   Using variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-273)

This section explains how to use variables, including the normal scalar variable syntax ${var}, how to use variables in list and dictionary contexts like @{var} and &{var}, respectively, and how to use environment variables like %{var}. Different ways how to create variables are discussed in the subsequent sections.

Robot Framework variables, similarly as keywords, are case-insensitive, and also spaces and underscores are ignored. However, it is recommended to use capital letters with global variables (for example, ${PATH} or ${TWO WORDS}) and small letters with local variables that are only available in certain test cases or user keywords (for example, ${my var}). Much more importantly, though, case should be used consistently.

Variable name consists of the variable type identifier ($, @, &, %), curly braces ({, }) and the actual variable name between the braces. Unlike in some programming languages where similar variable syntax is used, curly braces are always mandatory. Variable names can basically have any characters between the curly braces. However, using only alphabetic characters from a to z, numbers, underscore and space is recommended, and it is even a requirement for using the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax).

[**Scalar variable syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-274)

The most common way to use variables in Robot Framework test data is using the scalar variable syntax like ${var}. When this syntax is used, the variable name is replaced with its value as-is. Most of the time variable values are strings, but variables can contain any object, including numbers, lists, dictionaries, or even custom objects.

The example below illustrates the usage of scalar variables. Assuming that the variables ${GREET} and ${NAME} are available and assigned to strings Hello and world, respectively, both the example test cases are equivalent.

**\*\*\* Test Cases \*\*\***

**Constants**

Log Hello

Log Hello, world!!

**Variables**

Log ${GREET}

Log ${GREET}, ${NAME}!!

When a scalar variable is used alone without any text or other variables around it, like in ${GREET} above, the variable is replaced with its value as-is and the value can be any object. If the variable is not used alone, like ${GREER}, ${NAME}!! above, its value is first converted into a string and then concatenated with the other data.

**Note**

Variable values are used as-is without conversions also when passing arguments to keywords using the [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments) syntax like argname=${var}.

The example below demonstrates the difference between having a variable in alone or with other content. First, let us assume that we have a variable ${STR} set to a string Hello, world! and ${OBJ} set to an instance of the following Python object:

**class** **MyObj**:

**def** \_\_str\_\_():

**return** "Hi, terra!"

With these two variables set, we then have the following test data:

**\*\*\* Test Cases \*\*\***

**Objects**

KW 1 ${STR}

KW 2 ${OBJ}

KW 3 I said "${STR}"

KW 4 You said "${OBJ}"

Finally, when this test data is executed, different keywords receive the arguments as explained below:

* *KW 1* gets a string Hello, world!
* *KW 2* gets an object stored to variable ${OBJ}
* *KW 3* gets a string I said "Hello, world!"
* *KW 4* gets a string You said "Hi, terra!"

**Note**

Converting variables to Unicode obviously fails if the variable cannot be represented as Unicode. This can happen, for example, if you try to use byte sequences as arguments to keywords so that you catenate the values together like ${byte1}${byte2}. A workaround is creating a variable that contains the whole value and using it alone in the cell (e.g. ${bytes}) because then the value is used as-is.

[**List variable syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-275)

When a variable is used as a scalar like ${EXAMPLE}, its value is be used as-is. If a variable value is a list or list-like, it is also possible to use it as a list variable like @{EXAMPLE}. In this case the list is expanded and individual items are passed in as separate arguments. This is easiest to explain with an example. Assuming that a variable @{USER} has value ['robot', 'secret'], the following two test cases are equivalent:

**\*\*\* Test Cases \*\*\***

**Constants**

Login robot secret

**List Variable**

Login @{USER}

Robot Framework stores its own variables in one internal storage and allows using them as scalars, lists or dictionaries. Using a variable as a list requires its value to be a Python list or list-like object. Robot Framework does not allow strings to be used as lists, but other iterable objects such as tuples or dictionaries are accepted.

Starting from Robot Framework 4.0, list expansion can be used in combination with [list item access](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#accessing-sequence-items) making these usages possible:

**\*\*\* Test Cases \*\*\***

**Nested container**

${nested} = Evaluate [['a', 'b', 'c'], {'key': ['x', 'y']}]

Log Many @{nested}[0] *# Logs 'a', 'b' and 'c'.*

Log Many @{nested}[1][key] *# Logs 'x' and 'y'.*

**Slice**

${items} = Create List first second third

Log Many @{items}[1:] *# Logs 'second' and 'third'.*

**Using list variables with other data**

It is possible to use list variables with other arguments, including other list variables.

**\*\*\* Test Cases \*\*\***

**Example**

Keyword @{LIST} more args

Keyword ${SCALAR} @{LIST} constant

Keyword @{LIST} @{ANOTHER} @{ONE MORE}

**Using list variables with settings**

List variables can be used only with some of the [settings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#all-available-settings-in-test-data). They can be used in arguments to imported libraries and variable files, but library and variable file names themselves cannot be list variables. Also with setups and teardowns list variable can not be used as the name of the keyword, but can be used in arguments. With tag related settings they can be used freely. Using scalar variables is possible in those places where list variables are not supported.

**\*\*\* Settings \*\*\***

**Library** **ExampleLibrary** @{LIB ARGS} *# This works*

**Library** ${LIBRARY} @{LIB ARGS} *# This works*

**Library** @{LIBRARY AND ARGS} *# This does not work*

**Suite Setup** Some Keyword @{KW ARGS} *# This works*

**Suite Setup** ${KEYWORD} @{KW ARGS} *# This works*

**Suite Setup** @{KEYWORD AND ARGS} *# This does not work*

**Default Tags** @{TAGS} *# This works*

[**Dictionary variable syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-276)

As discussed above, a variable containing a list can be used as a [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) to pass list items to a keyword as individual arguments. Similarly a variable containing a Python dictionary or a dictionary-like object can be used as a dictionary variable like &{EXAMPLE}. In practice this means that the dictionary is expanded and individual items are passed as [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments) to the keyword. Assuming that a variable &{USER} has value {'name': 'robot', 'password': 'secret'}, the following two test cases are equivalent.

**\*\*\* Test Cases \*\*\***

**Constants**

Login name=robot password=secret

**Dict Variable**

Login &{USER}

Starting from Robot Framework 4.0, dictionary expansion can be used in combination with [dictionary item access](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#accessing-individual-dictionary-items) making usages like &{nested}[key] possible.

**Using dictionary variables with other data**

It is possible to use dictionary variables with other arguments, including other dictionary variables. Because [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) requires positional arguments to be before named argument, dictionaries can only be followed by named arguments or other dictionaries.

**\*\*\* Test Cases \*\*\***

**Example**

Keyword &{DICT} named=arg

Keyword positional @{LIST} &{DICT}

Keyword &{DICT} &{ANOTHER} &{ONE MORE}

**Using dictionary variables with settings**

Dictionary variables cannot generally be used with settings. The only exception are imports, setups and teardowns where dictionaries can be used as arguments.

**\*\*\* Settings \*\*\***

**Library** **ExampleLibrary** &{LIB ARGS}

**Suite Setup** Some Keyword &{KW ARGS} named=arg

[**Accessing list and dictionary items**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-277)

It is possible to access items of subscriptable variables, e.g. lists and dictionaries, using special syntax like ${var}[item] or ${var}[nested][item]. Starting from Robot Framework 4.0, it is also possible to use item access together with [list expansion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-expansion) and [dictionary expansion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-expansion) by using syntax @{var}[item] and &{var}[item], respectively.

**Note**

Prior to Robot Framework 3.1 the normal item access syntax was @{var}[item] with lists and &{var}[item] with dictionaries. Robot Framework 3.1 introduced the generic ${var}[item] syntax along with some other nice enhancements and the old item access syntax was deprecated in Robot Framework 3.2.

**Accessing sequence items**

It is possible to access a certain item of a variable containing a [sequence](https://docs.python.org/3/glossary.html#term-sequence) (e.g. list, string or bytes) with the syntax ${var}[index], where index is the index of the selected value. Indices start from zero, negative indices can be used to access items from the end, and trying to access an item with too large an index causes an error. Indices are automatically converted to integers, and it is also possible to use variables as indices.

**\*\*\* Test Cases \*\*\***

**Positive index**

Login ${USER}[0] ${USER}[1]

Title Should Be Welcome ${USER}[0]!

**Negative index**

Keyword ${SEQUENCE}[-1]

**Index defined as variable**

Keyword ${SEQUENCE}[${INDEX}]

Sequence item access supports also the [same "slice" functionality as Python](https://docs.python.org/glossary.html#term-slice) with syntax like ${var}[1:]. With this syntax you do not get a single item but a slice of the original sequence. Same way as with Python you can specify the start index, the end index, and the step:

**\*\*\* Test Cases \*\*\***

**Start index**

Keyword ${SEQUENCE}[1:]

**End index**

Keyword ${SEQUENCE}[:4]

**Start and end**

Keyword ${SEQUENCE}[2:-1]

**Step**

Keyword ${SEQUENCE}[::2]

Keyword ${SEQUENCE}[1:-1:10]

**Note**

The slice syntax is new in Robot Framework 3.1. It was extended to work with [list expansion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-expansion) like @{var}[1:] in Robot Framework 4.0.

**Note**

Prior to Robot Framework 3.2, item and slice access was only supported with variables containing lists, tuples, or other objects considered list-like. Nowadays all sequences, including strings and bytes, are supported.

**Accessing individual dictionary items**

It is possible to access a certain value of a dictionary variable with the syntax ${NAME}[key], where key is the name of the selected value. Keys are considered to be strings, but non-strings keys can be used as variables. Dictionary values accessed in this manner can be used similarly as scalar variables.

If a key is a string, it is possible to access its value also using attribute access syntax ${NAME.key}. See [Creating dictionary variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-dictionary-variables) for more details about this syntax.

**\*\*\* Test Cases \*\*\***

**Dictionary variable item**

Login ${USER}[name] ${USER}[password]

Title Should Be Welcome ${USER}[name]!

**Key defined as variable**

Log Many ${DICT}[${KEY}] ${DICT}[${42}]

**Attribute access**

Login ${USER.name} ${USER.password}

Title Should Be Welcome ${USER.name}!

**Nested item access**

Also nested subscriptable variables can be accessed using the same item access syntax like ${var}[item1][item2]. This is especially useful when working with JSON data often returned by REST services. For example, if a variable ${DATA} contains [{'id': 1, 'name': 'Robot'}, {'id': 2, 'name': 'Mr. X'}], this tests would pass:

**\*\*\* Test Cases \*\*\***

**Nested item access**

Should Be Equal ${DATA}[0][name] Robot

Should Be Equal ${DATA}[1][id] ${2}

[**Environment variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-278)

Robot Framework allows using environment variables in the test data using the syntax %{ENV\_VAR\_NAME}. They are limited to string values. It is possible to specify a default value, that is used if the environment variable does not exists, by separating the variable name and the default value with an equal sign like %{ENV\_VAR\_NAME=default value}.

Environment variables set in the operating system before the test execution are available during it, and it is possible to create new ones with the keyword *Set Environment Variable* or delete existing ones with the keyword *Delete Environment Variable*, both available in the [OperatingSystem](https://robotframework.org/robotframework/latest/libraries/OperatingSystem.html) library. Because environment variables are global, environment variables set in one test case can be used in other test cases executed after it. However, changes to environment variables are not effective after the test execution.

**\*\*\* Test Cases \*\*\***

**Environment variables**

Log Current user: %{USER}

Run %{JAVA\_HOME}${/}javac

**Environment variables with defaults**

Set port %{APPLICATION\_PORT=8080}

**Note**

Support for specifying the default value is new in Robot Framework 3.2.

[**2.6.3   Creating variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-279)

Variables can spring into existence from different sources.

[**Variable section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-280)

The most common source for variables are Variable sections in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files) and [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files). Variable sections are convenient, because they allow creating variables in the same place as the rest of the test data, and the needed syntax is very simple. Their main disadvantages are that values are always strings and they cannot be created dynamically. If either of these is a problem, [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files) can be used instead.

**Creating scalar variables**

The simplest possible variable assignment is setting a string into a scalar variable. This is done by giving the variable name (including ${}) in the first column of the Variable section and the value in the second one. If the second column is empty, an empty string is set as a value. Also an already defined variable can be used in the value.

**\*\*\* Variables \*\*\***

${NAME} Robot Framework

${VERSION} 2.0

${ROBOT} ${NAME} ${VERSION}

It is also possible, but not obligatory, to use the equals sign = after the variable name to make assigning variables slightly more explicit.

**\*\*\* Variables \*\*\***

${NAME} = Robot Framework

${VERSION} = 2.0

If a scalar variable has a long value, it can be [split into multiple rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows) by using the ... syntax. By default rows are concatenated together using a space, but this can be changed by having SEPARATOR=<sep> as the first item.

**\*\*\* Variables \*\*\***

${EXAMPLE} This value is joined

... together with a space.

${MULTILINE} SEPARATOR=\n

... First line.

... Second line.

... Third line.

**Creating list variables**

Creating list variables is as easy as creating scalar variables. Again, the variable name is in the first column of the Variable section and values in the subsequent columns. A list variable can have any number of values, starting from zero, and if many values are needed, they can be [split into several rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows).

**\*\*\* Variables \*\*\***

@{NAMES} Matti Teppo

@{NAMES2} @{NAMES} Seppo

@{NOTHING}

@{MANY} one two three four

... five six seven

**Creating dictionary variables**

Dictionary variables can be created in the Variable section similarly as list variables. The difference is that items need to be created using name=value syntax or existing dictionary variables. If there are multiple items with same name, the last value has precedence. If a name contains a literal equal sign, it can be [escaped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) with a backslash like \=.

**\*\*\* Variables \*\*\***

&{USER 1} name=Matti address=xxx phone=123

&{USER 2} name=Teppo address=yyy phone=456

&{MANY} first=1 second=${2} ${3}=third

&{EVEN MORE} &{MANY} first=override empty=

... =empty key\=here=value

Dictionary variables have two extra properties compared to normal Python dictionaries. First of all, values of these dictionaries can be accessed like attributes, which means that it is possible to use [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) like ${VAR.key}. This only works if the key is a valid attribute name and does not match any normal attribute Python dictionaries have. For example, individual value &{USER}[name] can also be accessed like ${USER.name} (notice that $ is needed in this context), but using ${MANY.3} is not possible.

**Tip**

With nested dictionary variables keys are accessible like ${VAR.nested.key}. This eases working with nested data structures.

Another special property of dictionary variables is that they are ordered. This means that if these dictionaries are iterated, their items always come in the order they are defined. This can be useful if dictionaries are used as [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables) with [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) or otherwise. When a dictionary is used as a list variable, the actual value contains dictionary keys. For example, @{MANY} variable would have value ['first', 'second', 3].

[**Variable file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-281)

Variable files are the most powerful mechanism for creating different kind of variables. It is possible to assign variables to any object using them, and they also enable creating variables dynamically. The variable file syntax and taking variable files into use is explained in section [Resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files).

[**Setting variables in command line**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-282)

Variables can be set from the command line either individually with the --variable (-v) option or using a variable file with the --variablefile (-V) option. Variables set from the command line are globally available for all executed test data files, and they also override possible variables with the same names in the Variable section and in variable files imported in the test data.

The syntax for setting individual variables is --variable name:value, where name is the name of the variable without ${} and value is its value. Several variables can be set by using this option several times. Only scalar variables can be set using this syntax and they can only get string values.

--variable EXAMPLE:value

--variable HOST:localhost:7272 --variable USER:robot

In the examples above, variables are set so that

* ${EXAMPLE} gets the value value
* ${HOST} and ${USER} get the values localhost:7272 and robot

The basic syntax for taking [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files) into use from the command line is --variablefile path/to/variables.py, and [Taking variable files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-variable-files-into-use) section has more details. What variables actually are created depends on what variables there are in the referenced variable file.

If both variable files and individual variables are given from the command line, the latter have [higher priority](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-priorities-and-scopes).

[**Return values from keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-283)

Return values from keywords can also be set into variables. This allows communication between different keywords even in different test libraries.

Variables set in this manner are otherwise similar to any other variables, but they are available only in the [local scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#local-scope) where they are created. Thus it is not possible, for example, to set a variable like this in one test case and use it in another. This is because, in general, automated test cases should not depend on each other, and accidentally setting a variable that is used elsewhere could cause hard-to-debug errors. If there is a genuine need for setting a variable in one test case and using it in another, it is possible to use [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords as explained in the next section.

**Assigning scalar variables**

Any value returned by a keyword can be assigned to a [scalar variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variable). As illustrated by the example below, the required syntax is very simple:

**\*\*\* Test Cases \*\*\***

**Returning**

${x} = Get X an argument

Log We got ${x}!

In the above example the value returned by the *Get X* keyword is first set into the variable ${x} and then used by the *Log* keyword. Having the equals sign = after the variable name is not obligatory, but it makes the assignment more explicit. Creating local variables like this works both in test case and user keyword level.

Notice that although a value is assigned to a scalar variable, it can be used as a [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) if it has a list-like value and as a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable) if it has a dictionary-like value.

**\*\*\* Test Cases \*\*\***

**Example**

${list} = Create List first second third

Length Should Be ${list} 3

Log Many @{list}

**Assigning list variables**

If a keyword returns a list or any list-like object, it is possible to assign it to a [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable):

**\*\*\* Test Cases \*\*\***

**Example**

@{list} = Create List first second third

Length Should Be ${list} 3

Log Many @{list}

Because all Robot Framework variables are stored in the same namespace, there is not much difference between assigning a value to a scalar variable or a list variable. This can be seen by comparing the last two examples above. The main differences are that when creating a list variable, Robot Framework automatically verifies that the value is a list or list-like, and the stored variable value will be a new list created from the return value. When assigning to a scalar variable, the return value is not verified and the stored value will be the exact same object that was returned.

**Assigning dictionary variables**

If a keyword returns a dictionary or any dictionary-like object, it is possible to assign it to a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable):

**\*\*\* Test Cases \*\*\***

**Example**

&{dict} = Create Dictionary first=1 second=${2} ${3}=third

Length Should Be ${dict} 3

Do Something &{dict}

Log ${dict.first}

Because all Robot Framework variables are stored in the same namespace, it would also be possible to assign a dictionary into a scalar variable and use it later as a dictionary when needed. There are, however, some actual benefits in creating a dictionary variable explicitly. First of all, Robot Framework verifies that the returned value is a dictionary or dictionary-like similarly as it verifies that list variables can only get a list-like value.

A bigger benefit is that the value is converted into a special dictionary that it uses also when [creating dictionary variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-dictionary-variables) in the Variable section. Values in these dictionaries can be accessed using attribute access like ${dict.first} in the above example. These dictionaries are also ordered, but if the original dictionary was not ordered, the resulting order is arbitrary.

**Assigning multiple variables**

If a keyword returns a list or a list-like object, it is possible to assign individual values into multiple scalar variables or into scalar variables and a list variable.

**\*\*\* Test Cases \*\*\***

**Assign multiple**

${a} ${b} ${c} = Get Three

${first} @{rest} = Get Three

@{before} ${last} = Get Three

${begin} @{middle} ${end} = Get Three

Assuming that the keyword *Get Three* returns a list [1, 2, 3], the following variables are created:

* ${a}, ${b} and ${c} with values 1, 2, and 3, respectively.
* ${first} with value 1, and @{rest} with value [2, 3].
* @{before} with value [1, 2] and ${last} with value 3.
* ${begin} with value 1, @{middle} with value [2] and ${end} with value 3.

It is an error if the returned list has more or less values than there are scalar variables to assign. Additionally, only one list variable is allowed and dictionary variables can only be assigned alone.

**Automatically logging assigned variable value**

To make it easier to understand what happens during execution, the beginning of value that is assigned is automatically logged. The default is to show 200 first characters, but this can be changed by using the --maxassignlength command line option when running tests. If the value is zero or negative, the whole assigned value is hidden.

--maxassignlength 1000

--maxassignlength 0

The reason the value is not logged fully is that it could be really big. If you always want to see a certain value fully, it is possible to use the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) *Log* keyword to log it after the assignment.

**Note**

The --maxassignlength option is new in Robot Framework 5.0.

[**Using *Set Test/Suite/Global Variable* keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-284)

The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library has keywords *Set Test Variable*, *Set Suite Variable* and *Set Global Variable* which can be used for setting variables dynamically during the test execution. If a variable already exists within the new scope, its value will be overwritten, and otherwise a new variable is created.

Variables set with *Set Test Variable* 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. It is an error to call *Set Test Variable* outside the scope of a test (e.g. in a Suite Setup or Teardown).

Variables set with *Set Suite Variable* 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 section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-section) in the test data file or importing them from [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files). Other test suites, including possible child test suites, will not see variables set with this keyword.

Variables set with *Set Global Variable* keyword are globally available in all test cases and suites executed after setting them. Setting variables with this keyword thus has the same effect as [creating from the command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) using the options --variable or --variablefile. Because this keyword can change variables everywhere, it should be used with care.

**Note**

*Set Test/Suite/Global Variable* keywords set named variables directly into [test, suite or global variable scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-scopes) and return nothing. On the other hand, another [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Set Variable* sets local variables using [return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords).

[**2.6.4   Built-in variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-285)

Robot Framework provides some built-in variables that are available automatically.

[**Operating-system variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-286)

Built-in variables related to the operating system ease making the test data operating-system-agnostic.

| *Available operating-system-related built-in variables* | |
| --- | --- |
| **Variable** | **Explanation** |
| ${CURDIR} | An absolute path to the directory where the test data file is located. This variable is case-sensitive. |
| ${TEMPDIR} | An absolute path to the system temporary directory. In UNIX-like systems this is typically */tmp*, and in Windows *c:\Documents and Settings\<user>\Local Settings\Temp*. |
| ${EXECDIR} | An absolute path to the directory where test execution was started from. |
| ${/} | The system directory path separator. / in UNIX-like systems and \ in Windows. |
| ${:} | The system path element separator. : in UNIX-like systems and ; in Windows. |
| ${\n} | The system line separator. \n in UNIX-like systems and \r\n in Windows. |

**\*\*\* Test Cases \*\*\***

**Example**

Create Binary File ${CURDIR}${/}input.data Some text here${\n}on two lines

Set Environment Variable CLASSPATH ${TEMPDIR}${:}${CURDIR}${/}foo.jar

[**Number variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-287)

The variable syntax can be used for creating both integers and floating point numbers, as illustrated in the example below. This is useful when a keyword expects to get an actual number, and not a string that just looks like a number, as an argument.

**\*\*\* Test Cases \*\*\***

**Example 1A**

Connect example.com 80 *# Connect gets two strings as arguments*

**Example 1B**

Connect example.com ${80} *# Connect gets a string and an integer*

**Example 2**

Do X ${3.14} ${-1e-4} *# Do X gets floating point numbers 3.14 and -0.0001*

It is possible to create integers also from binary, octal, and hexadecimal values using 0b, 0o and 0x prefixes, respectively. The syntax is case insensitive.

**\*\*\* Test Cases \*\*\***

**Example**

Should Be Equal ${0b1011} ${11}

Should Be Equal ${0o10} ${8}

Should Be Equal ${0xff} ${255}

Should Be Equal ${0B1010} ${0XA}

[**Boolean and None/null variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-288)

Also Boolean values and Python None can be created using the variable syntax similarly as numbers.

**\*\*\* Test Cases \*\*\***

**Boolean**

Set Status ${true} *# Set Status gets Boolean true as an argument*

Create Y something ${false} *# Create Y gets a string and Boolean false*

**None**

Do XYZ ${None} *# Do XYZ gets Python None as an argument*

These variables are case-insensitive, so for example ${True} and ${true} are equivalent.

[**Space and empty variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-289)

It is possible to create spaces and empty strings using variables ${SPACE} and ${EMPTY}, respectively. These variables are useful, for example, when there would otherwise be a need to [escape spaces or empty cells](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) with a backslash. If more than one space is needed, it is possible to use the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) like ${SPACE \* 5}. In the following example, *Should Be Equal* keyword gets identical arguments but those using variables are easier to understand than those using backslashes.

**\*\*\* Test Cases \*\*\***

**One space**

Should Be Equal ${SPACE} \ \

**Four spaces**

Should Be Equal ${SPACE \* 4} \ \ \ \ \

**Ten spaces**

Should Be Equal ${SPACE \* 10} \ \ \ \ \ \ \ \ \ \ \

**Quoted space**

Should Be Equal "${SPACE}" " "

**Quoted spaces**

Should Be Equal "${SPACE \* 2}" " \ "

**Empty**

Should Be Equal ${EMPTY} \

There is also an empty [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) @{EMPTY} and an empty [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable) &{EMPTY}. Because they have no content, they basically vanish when used somewhere in the test data. They are useful, for example, with [test templates](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates) when the [template keyword is used without arguments](https://groups.google.com/group/robotframework-users/browse_thread/thread/ccc9e1cd77870437/4577836fe946e7d5?lnk=gst&q=templates#4577836fe946e7d5) or when overriding list or dictionary variables in different scopes. Modifying the value of @{EMPTY} or &{EMPTY} is not possible.

**\*\*\* Test Cases \*\*\***

**Template**

[**Template**] Some keyword

@{EMPTY}

**Override**

Set Global Variable @{LIST} @{EMPTY}

Set Suite Variable &{DICT} &{EMPTY}

**Note**

${SPACE} represents the ASCII space (\x20) and [other spaces](http://jkorpela.fi/chars/spaces.html) should be specified using the [escape sequences](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) like \xA0 (NO-BREAK SPACE) and \u3000 (IDEOGRAPHIC SPACE).

[**Automatic variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-290)

Some automatic variables can also be used in the test data. These variables can have different values during the test execution and some of them are not even available all the time. Altering the value of these variables does not affect the original values, but some values can be changed dynamically using keywords from the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library.

| *Available automatic variables* | | |
| --- | --- | --- |
| **Variable** | **Explanation** | **Available** |
| ${TEST NAME} | The name of the current test case. | Test case |
| @{TEST TAGS} | Contains the tags of the current test case in alphabetical order. Can be modified dynamically using *Set Tags* and *Remove Tags* keywords. | Test case |
| ${TEST DOCUMENTATION} | The documentation of the current test case. Can be set dynamically using using *Set Test Documentation* keyword. | Test case |
| ${TEST STATUS} | The status of the current test case, either PASS or FAIL. | [Test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) |
| ${TEST MESSAGE} | The message of the current test case. | [Test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) |
| ${PREV TEST NAME} | The name of the previous test case, or an empty string if no tests have been executed yet. | Everywhere |
| ${PREV TEST STATUS} | The status of the previous test case: either PASS, FAIL, or an empty string when no tests have been executed. | Everywhere |
| ${PREV TEST MESSAGE} | The possible error message of the previous test case. | Everywhere |
| ${SUITE NAME} | The full name of the current test suite. | Everywhere |
| ${SUITE SOURCE} | An absolute path to the suite file or directory. | Everywhere |
| ${SUITE DOCUMENTATION} | The documentation of the current test suite. Can be set dynamically using using *Set Suite Documentation* keyword. | Everywhere |
| &{SUITE METADATA} | The free metadata of the current test suite. Can be set using *Set Suite Metadata* keyword. | Everywhere |
| ${SUITE STATUS} | The status of the current test suite, either PASS or FAIL. | [Suite teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) |
| ${SUITE MESSAGE} | The full message of the current test suite, including statistics. | [Suite teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) |
| ${KEYWORD STATUS} | The status of the current keyword, either PASS or FAIL. | [User keyword teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown) |
| ${KEYWORD MESSAGE} | The possible error message of the current keyword. | [User keyword teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown) |
| ${LOG LEVEL} | Current [log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). | Everywhere |
| ${OUTPUT FILE} | An absolute path to the [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file). | Everywhere |
| ${LOG FILE} | An absolute path to the [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) or string NONE when no log file is created. | Everywhere |
| ${REPORT FILE} | An absolute path to the [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) or string NONE when no report is created. | Everywhere |
| ${DEBUG FILE} | An absolute path to the [debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file) or string NONE when no debug file is created. | Everywhere |
| ${OUTPUT DIR} | An absolute path to the [output directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory). | Everywhere |
| &{OPTIONS} | A dictionary exposing command line options. The dictionary keys match the command line options and can be accessed both like ${OPTIONS}[key] and ${OPTIONS.key}. Available options:   * ${OPTIONS.exclude} (--exclude) * ${OPTIONS.include} (--include) * ${OPTIONS.skip} (--skip) * ${OPTIONS.skip\_on\_failure} (--skiponfailure)   New in RF 5.0. More options can be exposed later. | Everywhere |

Suite related variables ${SUITE SOURCE}, ${SUITE NAME}, ${SUITE DOCUMENTATION} and &{SUITE METADATA} as well as options related to command line options like ${LOG FILE} and &{OPTIONS} are available already when libraries and variable files are imported. Possible variables in these automatic variables are not yet resolved at the import time, though.

[**2.6.5   Variable priorities and scopes**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-291)

Variables coming from different sources have different priorities and are available in different scopes.

[**Variable priorities**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-292)

*Variables from the command line*

Variables [set in the command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) have the highest priority of all variables that can be set before the actual test execution starts. They override possible variables created in Variable sections in test case files, as well as in resource and variable files imported in the test data.

Individually set variables (--variable option) override the variables set using [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files) (--variablefile option). If you specify same individual variable multiple times, the one specified last will override earlier ones. This allows setting default values for variables in a [start-up script](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#start-up-script) and overriding them from the command line. Notice, though, that if multiple variable files have same variables, the ones in the file specified first have the highest priority.

*Variable section in a test case file*

Variables created using the [Variable section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-section) in a test case file are available for all the test cases in that file. These variables override possible variables with same names in imported resource and variable files.

Variables created in the Variable sections are available in all other sections in the file where they are created. This means that they can be used also in the Setting section, for example, for importing more variables from resource and variable files.

*Imported resource and variable files*

Variables imported from the [resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files) have the lowest priority of all variables created in the test data. Variables from resource files and variable files have the same priority. If several resource and/or variable file have same variables, the ones in the file imported first are taken into use.

If a resource file imports resource files or variable files, variables in its own Variable section have a higher priority than variables it imports. All these variables are available for files that import this resource file.

Note that variables imported from resource and variable files are not available in the Variable section of the file that imports them. This is due to the Variable section being processed before the Setting section where the resource files and variable files are imported.

*Variables set during test execution*

Variables set during the test execution either using [return values from keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) or [using Set Test/Suite/Global Variable keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-set-test-suite-global-variable-keywords) always override possible existing variables in the scope where they are set. In a sense they thus have the highest priority, but on the other hand they do not affect variables outside the scope they are defined.

*Built-in variables*

[Built-in variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables) like ${TEMPDIR} and ${TEST\_NAME} have the highest priority of all variables. They cannot be overridden using Variable section or from command line, but even they can be reset during the test execution. An exception to this rule are [number variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#number-variables), which are resolved dynamically if no variable is found otherwise. They can thus be overridden, but that is generally a bad idea. Additionally ${CURDIR} is special because it is replaced already during the test data processing time.

[**Variable scopes**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-293)

Depending on where and how they are created, variables can have a global, test suite, test case or local scope.

**Global scope**

Global variables are available everywhere in the test data. These variables are normally [set from the command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) with the --variable and --variablefile options, but it is also possible to create new global variables or change the existing ones with the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Set Global Variable* anywhere in the test data. Additionally also [built-in variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables) are global.

It is recommended to use capital letters with all global variables.

**Test suite scope**

Variables with the test suite scope are available anywhere in the test suite where they are defined or imported. They can be created in Variable sections, imported from [resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files), or set during the test execution using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Set Suite Variable*.

The test suite scope *is not recursive*, which means that variables available in a higher-level test suite *are not available* in lower-level suites. If necessary, [resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files) can be used for sharing variables.

Since these variables can be considered global in the test suite where they are used, it is recommended to use capital letters also with them.

**Test case scope**

Variables with the test case scope are visible in a test case and in all user keywords the test uses. Initially there are no variables in this scope, but it is possible to create them by using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Set Test Variable* anywhere in a test case. It is an error to call *Set Test Variable* outside the scope of a test (e.g. in a Suite Setup or Teardown).

Also variables in the test case scope are to some extend global. It is thus generally recommended to use capital letters with them too.

**Local scope**

Test cases and user keywords have a local variable scope that is not seen by other tests or keywords. Local variables can be created using [return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) from executed keywords and user keywords also get them as [arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments).

It is recommended to use lower-case letters with local variables.

[**2.6.6   Advanced variable features**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-294)

[**Extended variable syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-295)

Extended variable syntax allows accessing attributes of an object assigned to a variable (for example, ${object.attribute}) and even calling its methods (for example, ${obj.getName()}). It works both with scalar and list variables, but is mainly useful with the former

Extended variable syntax is a powerful feature, but it should be used with care. Accessing attributes is normally not a problem, on the contrary, because one variable containing an object with several attributes is often better than having several variables. On the other hand, calling methods, especially when they are used with arguments, can make the test data pretty complicated to understand. If that happens, it is recommended to move the code into a test library.

The most common usages of extended variable syntax are illustrated in the example below. First assume that we have the following [variable file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-file) and test case:

**class** **MyObject**:

**def** \_\_init\_\_(self, name):

self.name = name

**def** eat(self, what):

**return** '**%s** eats **%s**' % (self.name, what)

**def** \_\_str\_\_(self):

**return** self.name

OBJECT = MyObject('Robot')

DICTIONARY = {1: 'one', 2: 'two', 3: 'three'}

**\*\*\* Test Cases \*\*\***

**Example**

KW 1 ${OBJECT.name}

KW 2 ${OBJECT.eat('Cucumber')}

KW 3 ${DICTIONARY[2]}

When this test data is executed, the keywords get the arguments as explained below:

* *KW 1* gets string Robot
* *KW 2* gets string Robot eats Cucumber
* *KW 3* gets string two

The extended variable syntax is evaluated in the following order:

1. The variable is searched using the full variable name. The extended variable syntax is evaluated only if no matching variable is found.
2. The name of the base variable is created. The body of the name consists of all the characters after the opening { until the first occurrence of a character that is not an alphanumeric character or a space. For example, base variables of ${OBJECT.name} and ${DICTIONARY[2]}) are OBJECT and DICTIONARY, respectively.
3. A variable matching the body is searched. If there is no match, an exception is raised and the test case fails.
4. The expression inside the curly brackets is evaluated as a Python expression, so that the base variable name is replaced with its value. If the evaluation fails because of an invalid syntax or that the queried attribute does not exist, an exception is raised and the test fails.
5. The whole extended variable is replaced with the value returned from the evaluation.

Many standard Python objects, including strings and numbers, have methods that can be used with the extended variable syntax either explicitly or implicitly. Sometimes this can be really useful and reduce the need for setting temporary variables, but it is also easy to overuse it and create really cryptic test data. Following examples show few pretty good usages.

**\*\*\* Test Cases \*\*\***

**String**

${string} = Set Variable abc

Log ${string.upper()} *# Logs 'ABC'*

Log ${string \* 2} *# Logs 'abcabc'*

**Number**

${number} = Set Variable ${-2}

Log ${number \* 10} *# Logs -20*

Log ${number.\_\_abs\_\_()} *# Logs 2*

Note that even though abs(number) is recommended over number.\_\_abs\_\_() in normal Python code, using ${abs(number)} does not work. This is because the variable name must be in the beginning of the extended syntax. Using \_\_xxx\_\_ methods in the test data like this is already a bit questionable, and it is normally better to move this kind of logic into test libraries.

Extended variable syntax works also in [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) context. If, for example, an object assigned to a variable ${EXTENDED} has an attribute attribute that contains a list as a value, it can be used as a list variable @{EXTENDED.attribute}.

[**Extended variable assignment**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-296)

It is possible to set attributes of objects stored to scalar variables using [keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) and a variation of the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax). Assuming we have variable ${OBJECT} from the previous examples, attributes could be set to it like in the example below.

**\*\*\* Test Cases \*\*\***

**Example**

${OBJECT.name} = Set Variable New name

${OBJECT.new\_attr} = Set Variable New attribute

The extended variable assignment syntax is evaluated using the following rules:

1. The assigned variable must be a scalar variable and have at least one dot. Otherwise the extended assignment syntax is not used and the variable is assigned normally.
2. If there exists a variable with the full name (e.g. ${OBJECT.name} in the example above) that variable will be assigned a new value and the extended syntax is not used.
3. The name of the base variable is created. The body of the name consists of all the characters between the opening ${ and the last dot, for example, OBJECT in ${OBJECT.name} and foo.bar in ${foo.bar.zap}. As the second example illustrates, the base name may contain normal extended variable syntax.
4. The name of the attribute to set is created by taking all the characters between the last dot and the closing }, for example, name in ${OBJECT.name}. If the name does not start with a letter or underscore and contain only these characters and numbers, the attribute is considered invalid and the extended syntax is not used. A new variable with the full name is created instead.
5. A variable matching the base name is searched. If no variable is found, the extended syntax is not used and, instead, a new variable is created using the full variable name.
6. If the found variable is a string or a number, the extended syntax is ignored and a new variable created using the full name. This is done because you cannot add new attributes to Python strings or numbers, and this way the new syntax is also less backwards-incompatible.
7. If all the previous rules match, the attribute is set to the base variable. If setting fails for any reason, an exception is raised and the test fails.

**Note**

Unlike when assigning variables normally using [return values from keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords), changes to variables done using the extended assign syntax are not limited to the current scope. Because no new variable is created but instead the state of an existing variable is changed, all tests and keywords that see that variable will also see the changes.

[**Variables inside variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-297)

Variables are allowed also inside variables, and when this syntax is used, variables are resolved from the inside out. For example, if you have a variable ${var${x}}, then ${x} is resolved first. If it has the value name, the final value is then the value of the variable ${varname}. There can be several nested variables, but resolving the outermost fails, if any of them does not exist.

In the example below, *Do X* gets the value ${JOHN HOME} or ${JANE HOME}, depending on if *Get Name* returns john or jane. If it returns something else, resolving ${${name} HOME} fails.

**\*\*\* Variables \*\*\***

${JOHN HOME} /home/john

${JANE HOME} /home/jane

**\*\*\* Test Cases \*\*\***

**Example**

${name} = Get Name

Do X ${${name} HOME}

[**Inline Python evaluation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-298)

Variable syntax can also be used for evaluating Python expressions. The basic syntax is ${{expression}} i.e. there are double curly braces around the expression. The expression can be any valid Python expression such as ${{1 + 2}} or ${{['a', 'list']}}. Spaces around the expression are allowed, so also ${{ 1 + 2 }} and ${{ ['a', 'list'] }} are valid. In addition to using normal [scalar variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variables), also [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables) and [dictionary variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variables) support @{{expression}} and &{{expression}} syntax, respectively.

Main usages for this pretty advanced functionality are:

* Evaluating Python expressions involving Robot Framework's variables (${{len('${var}') > 3}}, ${{$var[0] if $var is not None else None}}).
* Creating values that are not Python base types (${{decimal.Decimal('0.11')}}, ${{datetime.date(2019, 11, 5)}}).
* Creating values dynamically (${{random.randint(0, 100)}}, ${{datetime.date.today()}}).
* Constructing collections, especially nested collections (${{[1, 2, 3, 4]}}, ${{ {'id': 1, 'name': 'Example', 'children': [7, 9]} }}).
* Accessing constants and other useful attributes in Python modules (${{math.pi}}, ${{platform.system()}}).

This is somewhat similar functionality than the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) discussed earlier. As the examples above illustrate, this syntax is even more powerful as it provides access to Python built-ins like len() and modules like math. In addition to being able to use variables like ${var} in the expressions (they are replaced before evaluation), variables are also available using the special $var syntax during evaluation. The whole expression syntax is explained in the [Evaluating expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#evaluating-expressions) appendix.

**Tip**

Instead of creating complicated expressions, it is often better to move the logic into a [custom library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries). That eases maintenance, makes test data easier to understand and can also enhance execution speed.

**Note**

The inline Python evaluation syntax is new in Robot Framework 3.2.

[**2.7   Creating user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-211)

Keyword sections are used to create new higher-level keywords by combining existing keywords together. These keywords are called *user keywords* to differentiate them from lowest level *library keywords* that are implemented in test libraries. The syntax for creating user keywords is very close to the syntax for creating test cases, which makes it easy to learn.

* [2.7.1   User keyword syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-syntax)
  + [Basic syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-syntax-2)
  + [Settings in the Keyword section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#settings-in-the-keyword-section)
* [2.7.2   User keyword name and documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-name-and-documentation)
* [2.7.3   User keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags)
* [2.7.4   User keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments)
  + [Positional arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments-with-user-keywords)
  + [Default values with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values-with-user-keywords)
  + [Variable number of arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments-with-user-keywords)
  + [Free named arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-user-keywords)
  + [Named-only arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-user-keywords)
* [2.7.5   Embedding arguments into keyword name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-name)
  + [Basic syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-syntax-3)
  + [Embedded arguments matching too much](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedded-arguments-matching-too-much)
  + [Using custom regular expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-custom-regular-expressions)
  + [Behavior-driven development example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#behavior-driven-development-example)
* [2.7.6   User keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-return-values)
  + [Using RETURN statement](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-return-statement)
  + [Using *[Return]* setting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-return-setting)
  + [Using special keywords to return](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-special-keywords-to-return)
* [2.7.7   User keyword teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown)

[**2.7.1   User keyword syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-299)

[**Basic syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-300)

In many ways, the overall user keyword syntax is identical to the [test case syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-syntax). User keywords are created in Keyword sections which differ from Test Case sections only by the name that is used to identify them. User keyword names are in the first column similarly as test cases names. Also user keywords are created from keywords, either from keywords in test libraries or other user keywords. Keyword names are normally in the second column, but when setting variables from keyword return values, they are in the subsequent columns.

**\*\*\* Keywords \*\*\***

**Open Login Page**

Open Browser http://host/login.html

Title Should Be Login Page

**Title Should Start With**

[**Arguments**] ${expected}

${title} = Get Title

Should Start With ${title} ${expected}

Most user keywords take some arguments. This important feature is used already in the second example above, and it is explained in detail [later in this section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments), similarly as [user keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-return-values).

User keywords can be created in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files), [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files), and [test suite initialization files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files). Keywords created in resource files are available for files using them, whereas other keywords are only available in the files where they are created.

[**Settings in the Keyword section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-301)

User keywords can have similar settings as [test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#settings-in-the-test-case-section), and they have the same square bracket syntax separating them from keyword names. All available settings are listed below and explained later in this section.

*[Documentation]*

Used for setting a [user keyword documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-documentation).

*[Tags]*

Sets [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags) for the keyword.

*[Arguments]*

Specifies [user keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments).

*[Return]*

Specifies [user keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-return-values). RETURN statement (new in RF 5.0) should be used instead.

*[Teardown]*

Specify [user keyword teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown).

*[Timeout]*

Sets the possible [user keyword timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-timeout). [Timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timeouts) are discussed in a section of their own.

**Note**

The format used above is recommended, but setting names are case-insensitive and spaces are allowed between brackets and the name. For example, [ TAGS ]:setting is valid.

[**2.7.2   User keyword name and documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-302)

The user keyword name is defined in the first column of the Keyword section. Of course, the name should be descriptive, and it is acceptable to have quite long keyword names. Actually, when creating use-case-like test cases, the highest-level keywords are often formulated as sentences or even paragraphs.

User keywords can have a documentation that is set with the *[Documentation]* setting. It supports same formatting, splitting to multiple lines, and other features as [test case documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation). This setting documents the user keyword in the test data. It is also shown in a more formal keyword documentation, which the [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) tool can create from [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files). Finally, the first logical row of the documentation, until the first empty row, is shown as a keyword documentation in [test logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file).

**\*\*\* Keywords \*\*\***

**One line documentation**

[**Documentation**] One line documentation.

No Operation

**Multiline documentation**

[**Documentation**] The first line creates the short doc.

...

... This is the body of the documentation.

... It is not shown in Libdoc outputs but only

... the short doc is shown in logs.

No Operation

**Short documentation in multiple lines**

[**Documentation**] If the short doc gets longer, it can span

... multiple physical lines.

...

... The body is separated from the short doc with

... an empty line.

No Operation

Sometimes keywords need to be removed, replaced with new ones, or deprecated for other reasons. User keywords can be marked deprecated by starting the documentation with \*DEPRECATED\*, which will cause a warning when the keyword is used. For more information, see the [Deprecating keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#deprecating-keywords) section.

**Note**

Prior to Robot Framework 3.1, the short documentation contained only the first physical line of the keyword documentation.

[**2.7.3   User keyword tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-303)

Both user keywords and [library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) can have tags. User keyword tags can be set with *[Tags]* setting similarly as [test case tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-tags), but possible *Force Tags* and *Default Tags* setting do not affect them. Additionally keyword tags can be specified on the last line of the documentation with Tags: prefix and separated by a comma. For example, following two keywords would both get same three tags.

**\*\*\* Keywords \*\*\***

**Settings tags using separate setting**

[**Tags**] my fine tags

No Operation

**Settings tags using documentation**

[**Documentation**] I have documentation. And my documentation has tags.

... Tags: my, fine, tags

No Operation

Keyword tags are shown in logs and in documentation generated by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc), where the keywords can also be searched based on tags. The [--removekeywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords) and [--flattenkeywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#flattening-keywords) commandline options also support selecting keywords by tag, and new usages for keywords tags are possibly added in later releases.

Similarly as with [test case tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-tags), user keyword tags with robot- and robot: prefixes are [reserved](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reserved-tags) for special features by Robot Framework itself. Users should thus not use any tag with these prefixes unless actually activating the special functionality.

[**2.7.4   User keyword arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-304)

Most user keywords need to take some arguments. The syntax for specifying them is probably the most complicated feature normally needed with Robot Framework, but even that is relatively easy, particularly in most common cases. Arguments are normally specified with the *[Arguments]* setting, and argument names use the same syntax as [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables), for example ${arg}.

[**Positional arguments with user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-305)

The simplest way to specify arguments (apart from not having them at all) is using only positional arguments. In most cases, this is all that is needed.

The syntax is such that first the *[Arguments]* setting is given and then argument names are defined in the subsequent cells. Each argument is in its own cell, using the same syntax as with variables. The keyword must be used with as many arguments as there are argument names in its signature. The actual argument names do not matter to the framework, but from users' perspective they should be as descriptive as possible. It is recommended to use lower-case letters in variable names, either as ${my\_arg}, ${my arg} or ${myArg}.

**\*\*\* Keywords \*\*\***

**One Argument**

[**Arguments**] ${arg\_name}

Log Got argument ${arg\_name}

**Three Arguments**

[**Arguments**] ${arg1} ${arg2} ${arg3}

Log 1st argument: ${arg1}

Log 2nd argument: ${arg2}

Log 3rd argument: ${arg3}

[**Default values with user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-306)

When creating user keywords, positional arguments are sufficient in most situations. It is, however, sometimes useful that keywords have [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values) for some or all of their arguments. Also user keywords support default values, and the needed new syntax does not add very much to the already discussed basic syntax.

In short, default values are added to arguments, so that first there is the equals sign (=) and then the value, for example ${arg}=default. There can be many arguments with defaults, but they all must be given after the normal positional arguments. The default value can contain a [variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) created on [test, suite or global scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-priorities-and-scopes), but local variables of the keyword executor cannot be used. Default value can also be defined based on earlier arguments accepted by the keyword.

**Note**

The syntax for default values is space sensitive. Spaces before the = sign are not allowed, and possible spaces after it are considered part of the default value itself.

**\*\*\* Keywords \*\*\***

**One Argument With Default Value**

[**Arguments**] ${arg}=default value

[**Documentation**] This keyword takes 0-1 arguments

Log Got argument ${arg}

**Two Arguments With Defaults**

[**Arguments**] ${arg1}=default 1 ${arg2}=${VARIABLE}

[**Documentation**] This keyword takes 0-2 arguments

Log 1st argument ${arg1}

Log 2nd argument ${arg2}

**One Required And One With Default**

[**Arguments**] ${required} ${optional}=default

[**Documentation**] This keyword takes 1-2 arguments

Log Required: ${required}

Log Optional: ${optional}

**Default Based On Earlier Argument**

[**Arguments**] ${a} ${b}=${a} ${c}=${a} and ${b}

Should Be Equal ${a} ${b}

Should Be Equal ${c} ${a} and ${b}

When a keyword accepts several arguments with default values and only some of them needs to be overridden, it is often handy to use the [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments) syntax. When this syntax is used with user keywords, the arguments are specified without the ${} decoration. For example, the second keyword above could be used like below and ${arg1} would still get its default value.

**\*\*\* Test Cases \*\*\***

**Example**

Two Arguments With Defaults arg2=new value

As all Pythonistas must have already noticed, the syntax for specifying default arguments is heavily inspired by Python syntax for function default values.

[**Variable number of arguments with user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-307)

Sometimes even default values are not enough and there is a need for a keyword accepting [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments). User keywords support also this feature. All that is needed is having [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) such as @{varargs} after possible positional arguments in the keyword signature. This syntax can be combined with the previously described default values, and at the end the list variable gets all the leftover arguments that do not match other arguments. The list variable can thus have any number of items, even zero.

**\*\*\* Keywords \*\*\***

**Any Number Of Arguments**

[**Arguments**] @{varargs}

Log Many @{varargs}

**One Or More Arguments**

[**Arguments**] ${required} @{rest}

Log Many ${required} @{rest}

**Required, Default, Varargs**

[**Arguments**] ${req} ${opt}=42 @{others}

Log Required: ${req}

Log Optional: ${opt}

Log Others:

FOR ${item} IN @{others}

Log ${item}

END

Notice that if the last keyword above is used with more than one argument, the second argument ${opt} always gets the given value instead of the default value. This happens even if the given value is empty. The last example also illustrates how a variable number of arguments accepted by a user keyword can be used in a [for loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops). This combination of two rather advanced functions can sometimes be very useful.

The keywords in the examples above could be used, for example, like this:

**\*\*\* Test Cases \*\*\***

**Varargs with user keywords**

Any Number Of Arguments

Any Number Of Arguments arg

Any Number Of Arguments arg1 arg2 arg3 arg4

One Or More Arguments required

One Or More Arguments arg1 arg2 arg3 arg4

Required, Default, Varargs required

Required, Default, Varargs required optional

Required, Default, Varargs arg1 arg2 arg3 arg4 arg5

Again, Pythonistas probably notice that the variable number of arguments syntax is very close to the one in Python.

[**Free named arguments with user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-308)

User keywords can also accept [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments) by having a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable) like &{named} as the absolutely last argument. When the keyword is called, this variable will get all [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments) that do not match any [positional argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments-with-user-keywords) or [named-only argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-user-keywords) in the keyword signature.

**\*\*\* Keywords \*\*\***

**Free Named Only**

[**Arguments**] &{named}

Log Many &{named}

**Positional And Free Named**

[**Arguments**] ${required} &{extra}

Log Many ${required} &{extra}

**Run Program**

[**Arguments**] @{args} &{config}

Run Process program.py @{args} &{config}

The last example above shows how to create a wrapper keyword that accepts any positional or named argument and passes them forward. See [free named argument examples](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-argument-examples) for a full example with same keyword.

Free named arguments support with user keywords works similarly as kwargs work in Python. In the signature and also when passing arguments forward, &{kwargs} is pretty much the same as Python's \*\*kwargs.

[**Named-only arguments with user keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-309)

Starting from Robot Framework 3.1, user keywords support [named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments) that are inspired by [Python 3 keyword-only arguments](https://www.python.org/dev/peps/pep-3102). This syntax is typically used by having normal arguments *after* [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments-with-user-keywords) (@{varargs}). If the keywords does not use varargs, it is possible to use just @{} to denote that the subsequent arguments are named-only:

**\*\*\* Keywords \*\*\***

**With Varargs**

[**Arguments**] @{varargs} ${named}

Log Many @{varargs} ${named}

**Without Varargs**

[**Arguments**] @{} ${first} ${second}

Log Many ${first} ${second}

Named-only arguments can be used together with [positional arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments-with-user-keywords) as well as with [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-user-keywords). When using free named arguments, they must be last:

**\*\*\* Keywords \*\*\***

**With Positional**

[**Arguments**] ${positional} @{} ${named}

Log Many ${positional} ${named}

**With Free Named**

[**Arguments**] @{varargs} ${named only} &{free named}

Log Many @{varargs} ${named only} &{free named}

When passing named-only arguments to keywords, their order does not matter other than they must follow possible positional arguments. The keywords above could be used, for example, like this:

**\*\*\* Test Cases \*\*\***

**Example**

With Varargs named=value

With Varargs positional second positional named=foobar

Without Varargs first=1 second=2

Without Varargs second=toka first=eka

With Positional foo named=bar

With Positional named=2 positional=1

With Free Named positional named only=value x=1 y=2

With Free Named foo=a bar=b named only=c quux=d

Named-only arguments can have default values similarly as [normal user keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values-with-user-keywords). A minor difference is that the order of arguments with and without default values is not important.

**\*\*\* Keywords \*\*\***

**With Default**

[**Arguments**] @{} ${named}=default

Log Many ${named}

**With And Without Defaults**

[**Arguments**] @{} ${optional}=default ${mandatory} ${mandatory 2} ${optional 2}=default 2 ${mandatory 3}

Log Many ${optional} ${mandatory} ${mandatory 2} ${optional 2} ${mandatory 3}

[**2.7.5   Embedding arguments into keyword name**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-310)

Robot Framework has also another approach to pass arguments to user keywords than specifying them in cells after the keyword name as explained in the previous section. This method is based on embedding the arguments directly into the keyword name, and its main benefit is making it easier to use real and clear sentences as keywords.

[**Basic syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-311)

It has always been possible to use keywords like *Select dog from list* and *Selects cat from list*, but all such keywords must have been implemented separately. The idea of embedding arguments into the keyword name is that all you need is a keyword with name like *Select ${animal} from list*.

**\*\*\* Keywords \*\*\***

**Select** ${animal} **from list**

Open Page Pet Selection

Select Item From List animal\_list ${animal}

Keywords using embedded arguments cannot take any "normal" arguments (specified with *[Arguments]* setting) but otherwise they are created just like other user keywords. The arguments used in the name will naturally be available inside the keyword and they have different value depending on how the keyword is called. For example, ${animal} in the previous has value dog if the keyword is used like *Select dog from list*. Obviously it is not mandatory to use all these arguments inside the keyword, and they can thus be used as wildcards.

These kind of keywords are also used the same way as other keywords except that spaces and underscores are not ignored in their names. They are, however, case-insensitive like other keywords. For example, the keyword in the example above could be used like *select x from list*, but not like *Select x fromlist*.

Embedded arguments do not support default values or variable number of arguments like normal arguments do. Using variables when calling these keywords is possible but that can reduce readability. Notice also that embedded arguments only work with user keywords.

[**Embedded arguments matching too much**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-312)

One tricky part in using embedded arguments is making sure that the values used when calling the keyword match the correct arguments. This is a problem especially if there are multiple arguments and characters separating them may also appear in the given values. For example, keyword *Select ${city} ${team}* does not work correctly if used with city containing two parts like *Select Los Angeles Lakers*.

An easy solution to this problem is quoting the arguments (e.g. *Select "${city}" "${team}"*) and using the keyword in quoted format (e.g. *Select "Los Angeles" "Lakers"*). This approach is not enough to resolve all this kind of conflicts, though, but it is still highly recommended because it makes arguments stand out from rest of the keyword. A more powerful but also more complicated solution, [using custom regular expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-custom-regular-expressions) when defining variables, is explained in the next section. Finally, if things get complicated, it might be a better idea to use normal positional arguments instead.

The problem of arguments matching too much occurs often when creating keywords that [ignore given/when/then/and/but prefixes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#ignoring-given-when-then-and-but-prefixes) . For example, *${name} goes home* matches *Given Janne goes home* so that ${name} gets value Given Janne. Quotes around the argument, like in *"${name}" goes home*, resolve this problem easily.

[**Using custom regular expressions**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-313)

When keywords with embedded arguments are called, the values are matched internally using [regular expressions](http://en.wikipedia.org/wiki/Regular_expression) (regexps for short). The default logic goes so that every argument in the name is replaced with a pattern .\*? that basically matches any string. This logic works fairly well normally, but as just discussed above, sometimes keywords [match more than intended](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedded-arguments-matching-too-much). Quoting or otherwise separating arguments from the other text can help but, for example, the test below fails because keyword *I execute "ls" with "-lh"* matches both of the defined keywords.

**\*\*\* Test Cases \*\*\***

**Example**

I execute "ls"

I execute "ls" with "-lh"

**\*\*\* Keywords \*\*\***

**I execute "**${cmd}**"**

Run Process ${cmd} shell=True

**I execute "**${cmd}**" with "**${opts}**"**

Run Process ${cmd} ${opts} shell=True

A solution to this problem is using a custom regular expression that makes sure that the keyword matches only what it should in that particular context. To be able to use this feature, and to fully understand the examples in this section, you need to understand at least the basics of the regular expression syntax.

A custom embedded argument regular expression is defined after the base name of the argument so that the argument and the regexp are separated with a colon. For example, an argument that should match only numbers can be defined like ${arg:\d+}. Using custom regular expressions is illustrated by the examples below.

**\*\*\* Test Cases \*\*\***

**Example**

I execute "ls"

I execute "ls" with "-lh"

I type 1 + 2

I type 53 - 11

Today is 2011-06-27

**\*\*\* Keywords \*\*\***

**I execute "**${cmd:[^"]+}**"**

Run Process ${cmd} shell=True

**I execute "**${cmd}**" with "**${opts}**"**

Run Process ${cmd} ${opts} shell=True

**I type** ${num1:\d+}${operator:[+-]}${num2:\d+}

Calculate ${num1} ${operator} ${num2}

**Today is** ${date:\d{4}**-\d{2}-\d{2}}**

Log ${date}

In the above example keyword *I execute "ls" with "-lh"* matches only *I execute "${cmd}" with "${opts}"*. That is guaranteed because the custom regular expression [^"]+ in *I execute "${cmd:[^"]}"* means that a matching argument cannot contain any quotes. In this case there is no need to add custom regexps to the other *I execute* variant.

**Tip**

If you quote arguments, using regular expression [^"]+ guarantees that the argument matches only until the first closing quote.

**Supported regular expression syntax**

Being implemented with Python, Robot Framework naturally uses Python's *re* module that has pretty standard [regular expressions syntax](http://docs.python.org/library/re.html). This syntax is otherwise fully supported with embedded arguments, but regexp extensions in format (?...) cannot be used. Notice also that matching embedded arguments is done case-insensitively. If the regular expression syntax is invalid, creating the keyword fails with an error visible in [test execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution).

**Escaping special characters**

Regular expressions use the backslash character (\) heavily both to escape characters that have a special meaning in regexps (e.g. \$) and to form special sequences (e.g. \d). Typically in Robot Framework data backslash characters [need to be escaped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) with another backslash, but that is not required in this context. If there is a need to have a literal backslash in the pattern, then the backslash must be escaped.

Possible lone opening and closing curly braces in the pattern must be escaped like ${open:\}} and ${close:\{}. If there are matching braces like ${two digits:\d{2}}, escaping is not needed. Escaping only opening or closing brace is not allowed.

**Warning**

Prior to Robot Framework 3.2 it was mandatory to escape all closing curly braces in the pattern like ${two digits:\d{2\}}. This syntax is unfortunately not supported by Robot Framework 3.2 or newer and keywords using it must be updated when upgrading.

**Using variables with custom embedded argument regular expressions**

Whenever custom embedded argument regular expressions are used, Robot Framework automatically enhances the specified regexps so that they match variables in addition to the text matching the pattern. This means that it is always possible to use variables with keywords having embedded arguments. For example, the following test case would pass using the keywords from the earlier example.

**\*\*\* Variables \*\*\***

${DATE} 2011-06-27

**\*\*\* Test Cases \*\*\***

**Example**

I type ${1} + ${2}

Today is ${DATE}

A drawback of variables automatically matching custom regular expressions is that it is possible that the value the keyword gets does not actually match the specified regexp. For example, variable ${DATE} in the above example could contain any value and *Today is ${DATE}* would still match the same keyword.

[**Behavior-driven development example**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-314)

The biggest benefit of having arguments as part of the keyword name is that it makes it easier to use higher-level sentence-like keywords when writing test cases in [behavior-driven style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#behavior-driven-style). The example below illustrates this. Notice also that prefixes *Given*, *When* and *Then* are [left out of the keyword definitions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#ignoring-given-when-then-and-but-prefixes).

**\*\*\* Test Cases \*\*\***

**Add two numbers**

*Given* I have Calculator open

*When* I add 2 and 40

*Then* result should be 42

**Add negative numbers**

*Given* I have Calculator open

*When* I add 1 and -2

*Then* result should be -1

**\*\*\* Keywords \*\*\***

**I have** ${program} **open**

Start Program ${program}

**I add** ${number 1} **and** ${number 2}

Input Number ${number 1}

Push Button +

Input Number ${number 2}

Push Button =

**Result should be** ${expected}

${result} = Get Result

Should Be Equal ${result} ${expected}

**Note**

Embedded arguments feature in Robot Framework is inspired by how *step definitions* are created in a popular BDD tool [Cucumber](http://cukes.info/).

[**2.7.6   User keyword return values**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-315)

Similarly as library keywords, also user keywords can return values. When using Robot Framework 5.0 or newer, the recommended approach is using the native RETURN statement. Old *[Return]* setting and [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Return From Keyword* and *Return From Keyword If* still work but they will be deprecated and removed in the future.

Regardless how values are returned, they can be [assigned to variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) in test cases and in other user keywords.

[**Using RETURN statement**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-316)

The recommended approach to return values is using the RETURN statement. It accepts optional return values and can be used with [IF](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if) and [inline IF](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-if) structures. Its usage is easiest explained with examples:

**\*\*\* Keywords \*\*\***

**Return One Value**

[**Arguments**] ${arg}

[**Documentation**] Return a value unconditionally.

... Notice that keywords after RETURN are not executed.

${value} = Convert To Upper Case ${arg}

RETURN ${value}

Fail Not executed

**Return Three Values**

[**Documentation**] Return multiple values.

RETURN a b c

**Conditional Return**

[**Arguments**] ${arg}

[**Documentation**] Return conditionally.

Log Before

IF ${arg} == 1

Log Returning!

RETURN

END

Log After

**Find Index**

[**Arguments**] ${test} ${items}

[**Documentation**] Advanced example involving FOR loop, inline IF and @{list} variable syntax.

FOR ${index} ${item} IN ENUMERATE @{items}

IF $item == $test RETURN ${index}

END

RETURN ${-1}

If you want to test the above examples yourself, you can use them with these test cases:

**\*\*\* Settings \*\*\***

**Library** **String**

**\*\*\* Test Cases \*\*\***

**One return value**

${ret} = Return One Value argument

Should Be Equal ${ret} ARGUMENT

**Multiple return values**

${a} ${b} ${c} = Return Three Values

Should Be Equal ${a}, ${b}, ${c} a, b, c

**Conditional return**

Conditional Return 1

Conditional Return 2

**Advanced**

@{list} = Create List foo bar baz

${index} = Find Index bar ${list}

Should Be Equal ${index} ${1}

${index} = Find Index non existing ${list}

Should Be Equal ${index} ${-1}

**Note**

RETURN syntax is case-sensitive similarly as [IF](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if) and [FOR](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for).

**Note**

RETURN is new in Robot Framework 5.0. Use approaches explained below if you need to support older versions.

[**Using *[Return]* setting**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-317)

The *[Return]* setting defines what the keyword should return after it has been executed. Although it is recommended to have it at the end of keyword where it logically belongs, its position does not affect how it is used.

An inherent limitation of the *[Return]* setting is that cannot be used conditionally. Thus only the first two earlier RETURN statement examples can be created using it.

**\*\*\* Keywords \*\*\***

**Return One Value**

[**Arguments**] ${arg}

${value} = Convert To Upper Case ${arg}

[**Return**] ${value}

**Return Three Values**

[**Return**] a b c

**Note**

The *[Return]* setting is effectively deprecated and the RETURN statement should be used unless there is a need to support also older versions than Robot Framework 5.0. There is no visible deprecation warning when using the setting yet, but it will be loudly deprecated and eventually removed in the future.

[**Using special keywords to return**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-318)

[BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Return From Keyword* and *Return From Keyword If* allow returning from a user keyword conditionally in the middle of the keyword. Both of them also accept optional return values that are handled exactly like with the RETURN statement and the *[Return]* setting discussed above.

The introduction of the RETURN statement makes these keywords redundant. Examples below contain same keywords as earlier RETURN examples but these ones are more verbose:

**\*\*\* Keywords \*\*\***

**Return One Value**

[**Arguments**] ${arg}

${value} = Convert To Upper Case ${arg}

Return From Keyword ${value}

Fail Not executed

**Return Three Values**

Return From Keyword a b c

**Conditional Return**

[**Arguments**] ${arg}

Log Before

IF ${arg} == 1

Log Returning!

Return From Keyword

END

Log After

**Find Index**

[**Arguments**] ${test} ${items}

FOR ${index} ${item} IN ENUMERATE @{items}

Return From Keyword If $item == $test ${index}

END

Return From Keyword ${-1}

**Note**

These keywords are effectively deprecated and the RETURN statement should be used unless there is a need to support also older versions than Robot Framework 5.0. There is no visible deprecation warning when using these keywords yet, but they will be loudly deprecated and eventually removed in the future.

[**2.7.7   User keyword teardown**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-319)

User keywords may have a teardown defined using *[Teardown]* setting.

Keyword teardown works much in the same way as a [test case teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). Most importantly, the teardown is always a single keyword, although it can be another user keyword, and it gets executed also when the user keyword fails. In addition, all steps of the teardown are executed even if one of them fails. However, a failure in keyword teardown will fail the test case and subsequent steps in the test are not run. The name of the keyword to be executed as a teardown can also be a variable.

**\*\*\* Keywords \*\*\***

**With Teardown**

Do Something

[**Teardown**] Log keyword teardown

**Using variables**

[**Documentation**] Teardown given as variable

Do Something

[**Teardown**] ${TEARDOWN}

[**2.8   Resource and variable files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-212)

User keywords and variables in [test case files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files) and [test suite initialization files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#initialization-files) can only be used in files where they are created, but *resource files* provide a mechanism for sharing them. The high level syntax for creating resource files is exactly the same as when creating test case files and [supported file formats](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats) are the same as well. The main difference is that resource files cannot have tests.

*Variable files* provide a powerful mechanism for creating and sharing variables. For example, they allow values other than strings and enable creating variables dynamically. Their flexibility comes from the fact that they are created using Python or YAML, which also makes them somewhat more complicated than [Variable sections](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-sections).

* [2.8.1   Resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files)
  + [Taking resource files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-resource-files-into-use)
  + [Resource file structure](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-file-structure)
  + [Documenting resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-resource-files)
  + [Example resource file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#example-resource-file)
* [2.8.2   Variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files)
  + [Taking variable files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-variable-files-into-use)
  + [Getting variables directly from a module](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-directly-from-a-module)
  + [Getting variables from a special function](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-from-a-special-function)
  + [Implementing variable file as a class](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implementing-variable-file-as-a-class)
  + [Variable file as YAML](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-file-as-yaml)

[**2.8.1   Resource files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-320)

[**Taking resource files into use**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-321)

Resource files are imported using the *Resource* setting in the Settings section. The path to the resource file is given as an argument to the setting. When using the [plain text format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats) for creating resource files, it is possible to use the normal *.robot* extension but the dedicated *.resource* extension is recommended to separate resource files from test case files.

If the path is given in an absolute format, it is used directly. In other cases, the resource file is first searched relatively to the directory where the importing file is located. If the file is not found there, it is then searched from the directories in Python's [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path). The path can contain variables, and it is recommended to use them to make paths system-independent (for example, *${RESOURCES}/login.resource* or *${RESOURCE\_PATH}*). Additionally, forward slashes (/) in the path are automatically changed to backslashes (\) on Windows.

**\*\*\* Settings \*\*\***

**Resource** **example.resource**

**Resource** **../data/resources.robot**

**Resource** ${RESOURCES}**/common.resource**

The user keywords and variables defined in a resource file are available in the file that takes that resource file into use. Similarly available are also all keywords and variables from the libraries, resource files and variable files imported by the said resource file.

**Note**

The *.resource* extension is new in Robot Framework 3.1.

[**Resource file structure**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-322)

The higher-level structure of resource files is the same as that of test case files otherwise, but, of course, they cannot contain Test Case sections. Additionally, the Setting section in resource files can contain only import settings (*Library*, *Resource*, *Variables*) and *Documentation*. The Variable section and Keyword section are used exactly the same way as in test case files.

If several resource files have a user keyword with the same name, they must be used so that the [keyword name is prefixed with the resource file name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-keywords-with-same-names) without the extension (for example, *myresources.Some Keyword* and *common.Some Keyword*). Moreover, if several resource files contain the same variable, the one that is imported first is taken into use.

[**Documenting resource files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-323)

Keywords created in a resource file can be [documented](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-name-and-documentation) using *[Documentation]* setting. The resource file itself can have *Documentation* in the Setting section similarly as [test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation).

Both [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) and [RIDE](https://github.com/robotframework/RIDE) use these documentations, and they are naturally available for anyone opening resource files. The first logical line of the documentation of a keyword, until the first empty line, is logged when the keyword is run, but otherwise resource file documentation is ignored during the test execution.

[**Example resource file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-324)

**\*\*\* Settings \*\*\***

**Documentation** An example resource file

**Library** **SeleniumLibrary**

**Resource** ${RESOURCES}**/common.resource**

**\*\*\* Variables \*\*\***

${HOST} localhost:7272

${LOGIN URL} http://${HOST}/

${WELCOME URL} http://${HOST}/welcome.html

${BROWSER} Firefox

**\*\*\* Keywords \*\*\***

**Open Login Page**

[**Documentation**] Opens browser to login page

Open Browser ${LOGIN URL} ${BROWSER}

Title Should Be Login Page

**Input Name**

[**Arguments**] ${name}

Input Text username\_field ${name}

**Input Password**

[**Arguments**] ${password}

Input Text password\_field ${password}

[**2.8.2   Variable files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-325)

Variable files contain [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) that can be used in the test data. Variables can also be created using [Variable sections](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-sections) or [set from the command line](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line), but variable files allow creating them dynamically and also make it easy to create other variable values than strings.

Variable files are typically implemented as modules and there are two different approaches for creating variables:

[Getting variables directly from a module](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-directly-from-a-module)

Variables are specified as module attributes. In simple cases, the syntax is so simple that no real programming is needed. For example, MY\_VAR = 'my value' creates a variable ${MY\_VAR} with the specified text as its value. One limitation of this approach is that it does not allow using arguments.

[Getting variables from a special function](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-from-a-special-function)

Variable files can have a special get\_variables (or getVariables) method that returns variables as a mapping. Because the method can take arguments this approach is very flexible.

Alternatively variable files can be implemented as [classes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implementing-variable-file-as-a-class) that the framework will instantiate. Also in this case it is possible to create variables as attributes or get them dynamically from the get\_variables method. Variable files can also be created as [YAML files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-file-as-yaml).

[**Taking variable files into use**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-326)

**Setting section**

All test data files can import variable files using the *Variables* setting in the Setting section. Variable files are typically imported using a path to the file same way as [resource files are imported](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-resource-files-into-use) using the *Resource* setting. Similarly to resource files, the path to the imported variable file is considered relative to the directory where the importing file is, and if not found, it is searched from directories in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path). The path can also contain variables, and slashes are converted to backslashes on Windows.

Examples:

**\*\*\* Settings \*\*\***

**Variables** **myvariables.py**

**Variables** **../data/variables.py**

**Variables** ${RESOURCES}**/common.yaml**

Starting from Robot Framework 5.0, variable files implemented using Python can also be imported using the module name [similarly as libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-library-to-import). When using this approach, the module needs to be in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path).

Examples:

**\*\*\* Settings \*\*\***

**Variables** **myvariables**

**Variables** **rootmodule.Variables**

If a [variable file accepts arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-from-a-special-function), they are specified after the path or name of the variable file to import:

**\*\*\* Settings \*\*\***

**Variables** **arguments.py** arg1 ${ARG2}

**Variables** **arguments** argument

All variables from a variable file are available in the test data file that imports it. If several variable files are imported and they contain a variable with the same name, the one in the earliest imported file is taken into use. Additionally, variables created in Variable sections and set from the command line override variables from variable files.

**Command line**

Another way to take variable files into use is using the command line option --variablefile. Variable files are referenced using a path or module name similarly as when importing them using the *Variables* setting. Possible arguments are joined to the path with a colon (:):

--variablefile myvariables.py

--variablefile path/variables.py

--variablefile /absolute/path/common.py

--variablefile variablemodule

--variablefile arguments.py:arg1:arg2

--variablefile rootmodule.Variables:arg1:arg2

Variable files taken into use from the command line are also searched from the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path) similarly as variable files imported in the Setting section. Relative paths are considered relative to the directory where execution is started from.

If a variable file is given as an absolute Windows path, the colon after the drive letter is not considered a separator:

--variablefile C:\path\variables.py

It is also possible to use a semicolon (;) as an argument separator. This is useful if variable file arguments themselves contain colons, but requires surrounding the whole value with quotes on UNIX-like operating systems:

--variablefile C:\path\variables.py;D:\data.xls

--variablefile "myvariables.py;argument:with:colons"

Variables in variable files taken use on the command line are globally available in all test data files, similarly as [individual variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) set with the --variable option. If both --variablefile and --variable options are used and there are variables with same names, those that are set individually with --variable option take precedence.

[**Getting variables directly from a module**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-327)

**Basic syntax**

When variable files are taken into use, they are imported as Python modules and all their module level attributes that do not start with an underscore (\_) are, by default, considered to be variables. Because variable names are case-insensitive, both lower- and upper-case names are possible, but in general, capital letters are recommended for global variables and attributes.

VARIABLE = "An example string"

ANOTHER\_VARIABLE = "This is pretty easy!"

INTEGER = 42

STRINGS = ["one", "two", "kolme", "four"]

NUMBERS = [1, INTEGER, 3.14]

MAPPING = {"one": 1, "two": 2, "three": 3}

In the example above, variables ${VARIABLE}, ${ANOTHER VARIABLE}, and so on, are created. The first two variables are strings, the third one is an integer, then there are two lists, and the final value is a dictionary. All these variables can be used as a [scalar variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variable), lists and the dictionary also a [list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variable) like @{STRINGS} (in the dictionary's case that variable would only contain keys), and the dictionary also as a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable) like &{MAPPING}.

To make creating a list variable or a dictionary variable more explicit, it is possible to prefix the variable name with LIST\_\_ or DICT\_\_, respectively:

**from** **collections** **import** OrderedDict

LIST\_\_ANIMALS = ["cat", "dog"]

DICT\_\_FINNISH = OrderedDict([("cat", "kissa"), ("dog", "koira")])

These prefixes will not be part of the final variable name, but they cause Robot Framework to validate that the value actually is list-like or dictionary-like. With dictionaries the actual stored value is also turned into a special dictionary that is used also when [creating dictionary variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-dictionary-variables) in the Variable section. Values of these dictionaries are accessible as attributes like ${FINNISH.cat}. These dictionaries are also ordered, but preserving the source order requires also the original dictionary to be ordered.

The variables in both the examples above could be created also using the Variable section below.

**\*\*\* Variables \*\*\***

${VARIABLE} An example string

${ANOTHER VARIABLE} This is pretty easy!

${INTEGER} ${42}

@{STRINGS} one two kolme four

@{NUMBERS} ${1} ${INTEGER} ${3.14}

&{MAPPING} one=${1} two=${2} three=${3}

@{ANIMALS} cat dog

&{FINNISH} cat=kissa dog=koira

**Note**

Variables are not replaced in strings got from variable files. For example, VAR = "an ${example}" would create variable ${VAR} with a literal string value an ${example} regardless would variable ${example} exist or not.

**Using objects as values**

Variables in variable files are not limited to having only strings or other base types as values like Variable sections. Instead, their variables can contain any objects. In the example below, the variable ${MAPPING} contains a Python dictionary and also has two variables created from a custom object implemented in the same file.

MAPPING = {'one': 1, 'two': 2}

**class** **MyObject**:

**def** \_\_init\_\_(self, name):

self.name = name

OBJ1 = MyObject('John')

OBJ2 = MyObject('Jane')

**Creating variables dynamically**

Because variable files are created using a real programming language, they can have dynamic logic for setting variables.

**import** **os**

**import** **random**

**import** **time**

USER = os.getlogin() *# current login name*

RANDOM\_INT = random.randint(0, 10) *# random integer in range [0,10]*

CURRENT\_TIME = time.asctime() *# timestamp like 'Thu Apr 6 12:45:21 2006'*

**if** time.localtime()[3] > 12:

AFTERNOON = **True**

**else**:

AFTERNOON = **False**

The example above uses standard Python libraries to set different variables, but you can use your own code to construct the values. The example below illustrates the concept, but similarly, your code could read the data from a database, from an external file or even ask it from the user.

**import** **math**

**def** get\_area(diameter):

radius = diameter / 2

area = math.pi \* radius \* radius

**return** area

AREA1 = get\_area(1)

AREA2 = get\_area(2)

**Selecting which variables to include**

When Robot Framework processes variable files, all their attributes that do not start with an underscore are expected to be variables. This means that even functions or classes created in the variable file or imported from elsewhere are considered variables. For example, the last example would contain the variables ${math} and ${get\_area} in addition to ${AREA1} and ${AREA2}.

Normally the extra variables do not cause problems, but they could override some other variables and cause hard-to-debug errors. One possibility to ignore other attributes is prefixing them with an underscore:

**import** **math** **as** **\_math**

**def** \_get\_area(diameter):

radius = diameter / 2.0

area = \_math.pi \* radius \* radius

**return** area

AREA1 = \_get\_area(1)

AREA2 = \_get\_area(2)

If there is a large number of other attributes, instead of prefixing them all, it is often easier to use a special attribute \_\_all\_\_ and give it a list of attribute names to be processed as variables.

**import** **math**

\_\_all\_\_ = ['AREA1', 'AREA2']

**def** get\_area(diameter):

radius = diameter / 2.0

area = math.pi \* radius \* radius

**return** area

AREA1 = get\_area(1)

AREA2 = get\_area(2)

**Note**

The \_\_all\_\_ attribute is also, and originally, used by Python to decide which attributes to import when using the syntax from modulename import \*.

The third option to select what variables are actually created is using a special get\_variables function discussed below.

[**Getting variables from a special function**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-328)

An alternative approach for getting variables is having a special get\_variables function (also camelCase syntax getVariables is possible) in a variable file. If such a function exists, Robot Framework calls it and expects to receive variables as a Python dictionary with variable names as keys and variable values as values. Created variables can be used as scalars, lists, and dictionaries exactly like when [getting variables directly from a module](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-directly-from-a-module), and it is possible to use LIST\_\_ and DICT\_\_ prefixes to make creating list and dictionary variables more explicit. The example below is functionally identical to the first example related to [getting variables directly from a module](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-variables-directly-from-a-module).

**def** get\_variables():

variables = {"VARIABLE ": "An example string",

"ANOTHER VARIABLE": "This is pretty easy!",

"INTEGER": 42,

"STRINGS": ["one", "two", "kolme", "four"],

"NUMBERS": [1, 42, 3.14],

"MAPPING": {"one": 1, "two": 2, "three": 3}}

**return** variables

get\_variables can also take arguments, which facilitates changing what variables actually are created. Arguments to the function are set just as any other arguments for a Python function. When [taking variable files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-variable-files-into-use) in the test data, arguments are specified in cells after the path to the variable file, and in the command line they are separated from the path with a colon or a semicolon.

The dummy example below shows how to use arguments with variable files. In a more realistic example, the argument could be a path to an external text file or database where to read variables from.

variables1 = {'scalar': 'Scalar variable',

'LIST\_\_list': ['List','variable']}

variables2 = {'scalar' : 'Some other value',

'LIST\_\_list': ['Some','other','value'],

'extra': 'variables1 does not have this at all'}

**def** get\_variables(arg):

**if** arg == 'one':

**return** variables1

**else**:

**return** variables2

[**Implementing variable file as a class**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-329)

It is possible to implement variables files also as a class.

**Implementation**

Because variable files are always imported using a file system path, the class must have the same name as the module it is located in.

The framework will create an instance of the class using no arguments and variables will be gotten from the instance. Similarly as with modules, variables can be defined as attributes directly in the instance or gotten from a special get\_variables method.

When variables are defined directly in an instance, all attributes containing callable values are ignored to avoid creating variables from possible methods the instance has. If you would actually need callable variables, you need to use other approaches to create variable files.

**Examples**

The first examples create variables from attributes. It creates variables ${VARIABLE} and @{LIST} from class attributes and ${ANOTHER VARIABLE} from an instance attribute.

**class** **StaticExample**:

variable = 'value'

LIST\_\_list = [1, 2, 3]

\_not\_variable = 'starts with an underscore'

**def** \_\_init\_\_(self):

self.another\_variable = 'another value'

The second examples utilizes dynamic approach for getting variables. It creates only one variable ${DYNAMIC VARIABLE}.

**class** **DynamicExample**:

**def** get\_variables(self, \*args):

**return** {'dynamic variable': ' '.join(args)}

[**Variable file as YAML**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-330)

Variable files can also be implemented as [YAML](http://yaml.org/) files. YAML is a data serialization language with a simple and human-friendly syntax. The following example demonstrates a simple YAML file:

**string**: Hello, world!

**integer**: 42

**list**:

- one

- two

**dict**:

**one**: yksi

**two**: kaksi

**with spaces**: kolme

**Note**

Using YAML files with Robot Framework requires [PyYAML](http://pyyaml.org/) module to be installed. If you have [pip](https://pip.pypa.io/) installed, you can install it simply by running pip install pyyaml.

YAML variable files must have either *.yaml* or *.yml* extension. Support for the *.yml* extension is new in Robot Framework 3.2.

YAML variable files can be used exactly like normal variable files from the command line using --variablefile option, in the Settings section using *Variables* setting, and dynamically using the *Import Variables* keyword.

If the above YAML file is imported, it will create exactly the same variables as this Variable section:

**\*\*\* Variables \*\*\***

${STRING} Hello, world!

${INTEGER} ${42}

@{LIST} one two

&{DICT} one=yksi two=kaksi

YAML files used as variable files must always be mappings in the top level. As the above example demonstrates, keys and values in the mapping become variable names and values, respectively. Variable values can be any data types supported by YAML syntax. If names or values contain non-ASCII characters, YAML variables files must be UTF-8 encoded.

Mappings used as values are automatically converted to special dictionaries that are used also when [creating dictionary variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-dictionary-variables) in the Variable section. Most importantly, values of these dictionaries are accessible as attributes like ${DICT.one}, assuming their names are valid as Python attribute names. If the name contains spaces or is otherwise not a valid attribute name, it is always possible to access dictionary values using syntax like ${DICT}[with spaces] syntax. The created dictionaries are also ordered, but unfortunately the original source order of in the YAML file is not preserved.

[**2.9   Control structures**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-213)

This section describes various structures that can be used to control the test execution flow. These structures are familiar from most programming languages and they allow conditional execution, repeatedly executing a block of keywords and fine-grained error handling. For readability reasons these structures should be used judiciously, and more complex use cases should be preferably implemented in [test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries).

* [2.9.1   FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops)
  + [Simple FOR loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-for-loop)
  + [Old FOR loop syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#old-for-loop-syntax)
  + [Nesting FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#nesting-for-loops)
  + [Using several loop variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-several-loop-variables)
  + [FOR-IN-RANGE loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-in-range-loop)
  + [FOR-IN-ENUMERATE loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-in-enumerate-loop)
  + [FOR-IN-ZIP loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-in-zip-loop)
  + [Dictionary iteration](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-iteration)
  + [Removing unnecessary keywords from outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-unnecessary-keywords-from-outputs)
  + [Repeating single keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#repeating-single-keyword)
* [2.9.2   WHILE loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#while-loops)
  + [Basic WHILE syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-while-syntax)
  + [Limiting WHILE loop iterations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-while-loop-iterations)
  + [Nesting WHILE loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#nesting-while-loops)
  + [Removing unnecessary keywords from outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-unnecessary-keywords-from-outputs-1)
* [2.9.3   Loop control using BREAK and CONTINUE](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#loop-control-using-break-and-continue)
* [2.9.4   IF/ELSE syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else-syntax)
  + [Basic IF syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-if-syntax)
  + [ELSE branches](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#else-branches)
  + [ELSE IF branches](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#else-if-branches)
  + [Inline IF](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-if-1)
  + [Nested IF structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#nested-if-structures)
  + [Other ways to execute keywords conditionally](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#other-ways-to-execute-keywords-conditionally)
* [2.9.5   TRY/EXCEPT syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#try-except-syntax)
  + [Catching exceptions with EXCEPT](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#catching-exceptions-with-except)
  + [Matching errors using patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#matching-errors-using-patterns)
  + [Capturing error message](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#capturing-error-message)
  + [Using ELSE to execute keywords when there are no errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-else-to-execute-keywords-when-there-are-no-errors)
  + [Using FINALLY to execute keywords regardless are there errors or not](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-finally-to-execute-keywords-regardless-are-there-errors-or-not)
  + [Other ways to handle errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#other-ways-to-handle-errors)

[**2.9.1   FOR loops**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-331)

Repeating same actions several times is quite a common need in test automation. With Robot Framework, test libraries can have any kind of loop constructs, and most of the time loops should be implemented in them. Robot Framework also has its own FOR loop syntax, which is useful, for example, when there is a need to repeat keywords from different libraries.

FOR loops can be used with both test cases and user keywords. Except for really simple cases, user keywords are better, because they hide the complexity introduced by FOR loops. The basic FOR loop syntax, FOR item IN sequence, is derived from Python, but similar syntax is supported also by various other programming languages.

[**Simple FOR loop**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-332)

In a normal FOR loop, one variable is assigned based on a list of values, one value per iteration. The syntax starts with FOR (case-sensitive) as a marker, then the loop variable, then a mandatory IN (case-sensitive) as a separator, and finally the values to iterate. These values can contain [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables), including [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables).

The keywords used in the FOR loop are on the following rows and the loop ends with END (case-sensitive) on its own row. Keywords inside the loop do not need to be indented, but that is highly recommended to make the syntax easier to read.

**\*\*\* Test Cases \*\*\***

**Example**

FOR ${animal} IN cat dog

Log ${animal}

Log 2nd keyword

END

Log Outside loop

**Second Example**

FOR ${var} IN one two ${3} four ${five}

... kuusi 7 eight nine ${last}

Log ${var}

END

The FOR loop in *Example* above is executed twice, so that first the loop variable ${animal} has the value cat and then dog. The loop consists of two *Log* keywords. In the second example, loop values are [split into two rows](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows) and the loop is run altogether ten times.

It is often convenient to use FOR loops with [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables). This is illustrated by the example below, where @{ELEMENTS} contains an arbitrarily long list of elements and keyword *Start Element* is used with all of them one by one.

**\*\*\* Test Cases \*\*\***

**Example**

FOR ${element} IN @{ELEMENTS}

Start Element ${element}

END

[**Old FOR loop syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-333)

Prior to Robot Framework 3.1, the FOR loop syntax was different than nowadays. The marker to start the loop was :FOR instead of FOR and loop contents needed to be explicitly marked with a backslash instead of using the END marker to end the loop. The first example above would look like this using the old syntax:

**\*\*\* Test Cases \*\*\***

**Example**

:FOR ${animal} IN cat dog

\ Log ${animal}

\ Log 2nd keyword

Log Outside loop

The old syntax was deprecated in Robot Framework 3.2 and the support for it was removed altogether in Robot Framework 4.0.

[**Nesting FOR loops**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-334)

Starting from Robot Framework 4.0, it is possible to use nested FOR loops simply by adding a loop inside another loop:

**\*\*\* Keywords \*\*\***

**Handle Table**

[**Arguments**] @{table}

FOR ${row} IN @{table}

FOR ${cell} IN @{row}

Handle Cell ${cell}

END

END

There can be multiple nesting levels and loops can also be combined with other control structures:

**\*\*\* Test Cases \*\*\***

**Multiple nesting levels**

FOR ${root} IN r1 r2

FOR ${child} IN c1 c2 c3

FOR ${grandchild} IN g1 g2

Log Many ${root} ${child} ${grandchild}

END

END

FOR ${sibling} IN s1 s2 s3

IF '${sibling}' != 's2'

Log Many ${root} ${sibling}

END

END

END

[**Using several loop variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-335)

It is possible to iterate over multiple values in one iteration by using multiple loop variables between the FOR and IN markers. There can be any number of loop variables, but the number of values must be evenly dividable by the number of variables. Each iteration consumes as many values as there are variables.

If there are lot of values to iterate, it is often convenient to organize them below the loop variables, as in the first loop of the example below:

**\*\*\* Test Cases \*\*\***

**Multiple loop variables**

FOR ${index} ${english} ${finnish} IN

... 1 cat kissa

... 2 dog koira

... 3 horse hevonen

Add Translation ${english} ${finnish} ${index}

END

FOR ${name} ${id} IN @{EMPLOYERS}

Create ${name} ${id}

END

[**FOR-IN-RANGE loop**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-336)

All FOR loops in the previous section iterated over a sequence. That is the most common use case, but sometimes it is convenient to have a loop that is executed a certain number of times. For this purpose Robot Framework has a special FOR index IN RANGE limit loop syntax that is derived from the similar Python idiom using the [built-in range() function](http://docs.python.org/library/functions.html#func-range).

Similarly as other FOR loops, the FOR-IN-RANGE loop starts with FOR that is followed by a loop variable. In this format there can be only one loop variable and it contains the current loop index. After the variable there must be IN RANGE marker (case-sensitive) that is followed by loop limits.

In the simplest case, only the upper limit of the loop is specified. In this case, loop indices start from zero and increase by one until, but excluding, the limit. It is also possible to give both the start and end limits. Then indices start from the start limit, but increase similarly as in the simple case. Finally, it is possible to give also the step value that specifies the increment to use. If the step is negative, it is used as decrement.

It is possible to use simple arithmetic such as addition and subtraction with the range limits. This is especially useful when the limits are specified with variables. Start, end and step are typically given as integers, but using float values is possible as well.

**\*\*\* Test Cases \*\*\***

**Only upper limit**

[**Documentation**] Loops over values from 0 to 9.

FOR ${index} IN RANGE 10

Log ${index}

END

**Start and end**

[**Documentation**] Loops over values from 1 to 10.

FOR ${index} IN RANGE 1 11

Log ${index}

END

**Also step given**

[**Documentation**] Loops over values 5, 15, and 25.

FOR ${index} IN RANGE 5 26 10

Log ${index}

END

**Negative step**

[**Documentation**] Loops over values 13, 3, and -7.

FOR ${index} IN RANGE 13 -13 -10

Log ${index}

END

**Arithmetic**

[**Documentation**] Arithmetic with variable.

FOR ${index} IN RANGE ${var} + 1

Log ${index}

END

**Float parameters**

[**Documentation**] Loops over values 3.14, 4.34, and 5.54.

FOR ${index} IN RANGE 3.14 6.09 1.2

Log ${index}

END

[**FOR-IN-ENUMERATE loop**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-337)

Sometimes it is useful to loop over a list and also keep track of your location inside the list. Robot Framework has a special FOR index ... IN ENUMERATE ... syntax for this situation. This syntax is derived from the [Python built-in enumerate() function](http://docs.python.org/library/functions.html#enumerate).

FOR-IN-ENUMERATE loops syntax is just like the regular FOR loop syntax, except that the separator between variables and values is IN ENUMERATE (case-sensitive). Typically they are used so that there is an additional index variable before any other loop-variables. By default the index has a value 0 on the first iteration, 1 on the second, and so on.

For example, the following two test cases do the same thing:

**\*\*\* Variables \*\*\***

@{LIST} a b c

**\*\*\* Test Cases \*\*\***

**Manage index manually**

${index} = Set Variable -1

FOR ${item} IN @{LIST}

${index} = Evaluate ${index} + 1

My Keyword ${index} ${item}

END

**FOR-IN-ENUMERATE**

FOR ${index} ${item} IN ENUMERATE @{LIST}

My Keyword ${index} ${item}

END

Starting from Robot Framework 4.0, it is possible to specify a custom start index by using start=<index> syntax as the last item of the FOR ... IN ENUMERATE ... header:

**\*\*\* Variables \*\*\***

@{LIST} a b c

${START} 10

**\*\*\* Test Cases \*\*\***

**FOR-IN-ENUMERATE with start**

FOR ${index} ${item} IN ENUMERATE @{LIST} start=1

My Keyword ${index} ${item}

END

**Start as variable**

FOR ${index} ${item} IN ENUMERATE @{LIST} start=${start}

My Keyword ${index} ${item}

END

The start=<index> syntax must be explicitly used in the FOR header and it cannot itself come from a variable. If the last actual item to enumerate would start with start=, it needs to be escaped like start\=.

Just like with regular FOR loops, you can loop over multiple values per loop iteration as long as the number of values in your list is evenly divisible by the number of loop-variables (excluding the index variable):

**\*\*\* Test Case \*\*\***

**FOR-IN-ENUMERATE with two values per iteration**

FOR ${index} ${en} ${fi} IN ENUMERATE

... cat kissa

... dog koira

... horse hevonen

Log "${en}" in English is "${fi}" in Finnish (index: ${index})

END

If you only use one loop variable with FOR-IN-ENUMERATE loops, that variable will become a Python tuple containing the index and the iterated value:

**\*\*\* Test Case \*\*\***

**FOR-IN-ENUMERATE with one loop variable**

FOR ${x} IN ENUMERATE @{LIST}

Length Should Be ${x} 2

Log Index is ${x}[0] and item is ${x}[1].

END

**Note**

FOR-IN-ENUMERATE loops with only one loop variable is a new feature in Robot Framework 3.2.

[**FOR-IN-ZIP loop**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-338)

Some tests build up several related lists, then loop over them together. Robot Framework has a shortcut for this case: FOR ... IN ZIP ..., which is derived from the [Python built-in zip() function](http://docs.python.org/library/functions.html#zip).

This may be easiest to show with an example:

**\*\*\* Variables \*\*\***

@{NUMBERS} ${1} ${2} ${5}

@{NAMES} one two five

**\*\*\* Test Cases \*\*\***

**Iterate over two lists manually**

${length}= Get Length ${NUMBERS}

FOR ${index} IN RANGE ${length}

Log Many ${NUMBERS}[${index}] ${NAMES}[${index}]

END

**FOR-IN-ZIP**

FOR ${number} ${name} IN ZIP ${NUMBERS} ${NAMES}

Log Many ${number} ${name}

END

As the example above illustrates, FOR-IN-ZIP loops require their own custom separator IN ZIP (case-sensitive) between loop variables and values. Values used with FOR-IN-ZIP loops must be lists or list-like objects. Looping will stop when the shortest list is exhausted.

Lists to iterate over must always be given either as [scalar variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variables) like ${items} or as [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables) like @{lists} that yield the actual iterated lists. The former approach is more common and it was already demonstrated above. The latter approach works like this:

**\*\*\* Variables \*\*\***

@{NUMBERS} ${1} ${2} ${5}

@{NAMES} one two five

@{LISTS} ${NUMBERS} ${NAMES}

**\*\*\* Test Cases \*\*\***

**FOR-IN-ZIP with lists from variable**

FOR ${number} ${name} IN ZIP @{LISTS}

Log Many ${number} ${name}

END

The number of lists to iterate over is not limited, but it must match the number of loop variables. Alternatively there can be just one loop variable that then becomes a Python tuple getting items from all lists.

**\*\*\* Variables \*\*\***

@{ABC} a b c

@{XYZ} x y z

@{NUM} 1 2 3 4 5

**\*\*\* Test Cases \*\*\***

**FOR-IN-ZIP with multiple lists**

FOR ${a} ${x} ${n} IN ZIP ${ABC} ${XYZ} ${NUM}

Log Many ${a} ${x} ${n}

END

**FOR-IN-ZIP with one variable**

FOR ${items} IN ZIP ${ABC} ${XYZ} ${NUM}

Length Should Be ${items} 3

Log Many ${items}[0] ${items}[1] ${items}[2]

END

If lists have an unequal number of items, the shortest list defines how many iterations there are and values at the end of longer lists are ignored. For example, the above examples loop only three times and values 4 and 5 in the ${NUM} list are ignored.

**Note**

Getting lists to iterate over from list variables and using just one loop variable are new features in Robot Framework 3.2.

[**Dictionary iteration**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-339)

Normal FOR loops and FOR-IN-ENUMERATE loops support iterating over keys and values in dictionaries. This syntax requires at least one of the loop values to be a [dictionary variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dictionary-variable). It is possible to use multiple dictionary variables and to give additional items in key=value syntax. Items are iterated in the order they are defined and if same key gets multiple values the last value will be used.

**\*\*\* Variables \*\*\***

&{DICT} a=1 b=2 c=3

**\*\*\* Test Cases \*\*\***

**Dictionary iteration with FOR loop**

FOR ${key} ${value} IN &{DICT}

Log Key is '${key}' and value is '${value}'.

END

**Dictionary iteration with FOR-IN-ENUMERATE loop**

FOR ${index} ${key} ${value} IN ENUMERATE &{DICT}

Log On round ${index} key is '${key}' and value is '${value}'.

END

**Multiple dictionaries and extra items in 'key=value' syntax**

&{more} = Create Dictionary e=5 f=6

FOR ${key} ${value} IN &{DICT} d=4 &{more} g=7

Log Key is '${key}' and value is '${value}'.

END

Typically it is easiest to use the dictionary iteration syntax so that keys and values get separate variables like in the above examples. With normal FOR loops it is also possible to use just a single variable that will become a tuple containing the key and the value. If only one variable is used with FOR-IN-ENUMERATE loops, it becomes a tuple containing the index, the key and the value. Two variables with FOR-IN-ENUMERATE loops means assigning the index to the first variable and making the second variable a tuple containing the key and the value.

**\*\*\* Test Cases \*\*\***

**One loop variable**

FOR ${item} IN &{DICT}

Log Key is '${item}[0]' and value is '${item}[1]'.

END

**One loop variable with FOR-IN-ENUMERATE**

FOR ${item} IN ENUMERATE &{DICT}

Log On round ${item}[0] key is '${item}[1]' and value is '${item}[2]'.

END

**Two loop variables with FOR-IN-ENUMERATE**

FOR ${index} ${item} IN ENUMERATE &{DICT}

Log On round ${index} key is '${item}[0]' and value is '${item}[1]'.

END

In addition to iterating over names and values in dictionaries, it is possible to iterate over keys and then possibly fetch the value based on it. This syntax requires using dictionaries as [list variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables):

**\*\*\* Test Cases \*\*\***

**Iterate over keys**

FOR ${key} IN @{DICT}

Log Key is '${key}' and value is '${DICT}[${key}]'.

END

**Note**

Iterating over keys and values in dictionaries is a new feature in Robot Framework 3.2. With earlier version it is possible to iterate over dictionary keys like the last example above demonstrates.

[**Removing unnecessary keywords from outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-340)

FOR loops with multiple iterations often create lots of output and considerably increase the size of the generated [output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) and [log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) files. It is possible to [remove or flatten unnecessary keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) using --removekeywords and --flattenkeywords command line options.

[**Repeating single keyword**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-341)

FOR loops can be excessive in situations where there is only a need to repeat a single keyword. In these cases it is often easier to use [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Repeat Keyword*. This keyword takes a keyword and how many times to repeat it as arguments. The times to repeat the keyword can have an optional postfix times or x to make the syntax easier to read.

**\*\*\* Test Cases \*\*\***

**Example**

Repeat Keyword 5 Some Keyword arg1 arg2

Repeat Keyword 42 times My Keyword

Repeat Keyword ${var} Another Keyword argument

[**2.9.2   WHILE loops**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-342)

WHILE loops combine features of [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) and [IF/ELSE structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else-structures). They specify a condition and repeat the loop body as long as the condition remains true. This can be utilised, for example, to repeat a nondeterministic sequence until the desired outcome happens, or in some cases they can be used as an alternative to [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops).

**Note**

WHILE loops are new in Robot Framework 5.0.

[**Basic WHILE syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-343)

**\*\*\* Test Cases \*\*\***

**Example**

${rc} = Set Variable 1

WHILE ${rc} != 0

${rc} = Keyword that returns zero on success

END

The WHILE loop condition is evaluated in Python so that Python builtins like len() are available and modules are imported automatically to support usages like math.pi \* math.pow(${radius}, 2) < 10. Normal variables like ${rc} in the above example are replaced before evaluation, but variables are also available in the evaluation namespace using the special $rc syntax. The latter approach is handy when the string representation of the variable cannot be used in the condition directly. For example, strings require quoting and multiline strings and string themselves containing quotes cause additional problems. See the [Evaluating expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#evaluating-expressions) appendix for more information and examples related to the evaluation syntax

[**Limiting WHILE loop iterations**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-344)

With WHILE loops, there is always a possibility to achieve an infinite loop, either by intention or by mistake. This happens when the loop condition never becomes false. While infinite loops have some utility in application programming, in automation an infinite loop is rarely a desired outcome. If such a loop occurs with Robot Framework, the execution must be forcefully stopped and no log or report can be created. For this reason, WHILE loops in Robot Framework have a default limit of 10 000 iterations. If the limit is exceeded, the loop fails.

The limit can be changed with the limit configuration parameter. Valid values are positive integers denoting iteration count and [time strings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format) like 10s or 1 hour 10 minutes denoting maximum iteration time. The limit can also be disabled altogether by using NONE (case-insensitive). All these options are illustrated by the examples below.

**\*\*\* Test Cases \*\*\***

**Limit as iteration count**

WHILE True limit=100

Log This is run 100 times.

END

**Limit as time**

WHILE True limit=10 seconds

Log This is run 10 seconds.

END

**No limit**

WHILE True limit=NONE

Log This must be forcefully stopped.

END

Keywords in a loop are not forcefully stopped if the limit is exceeded. Instead the loop is exited similarly as if the loop condition would have become false. A major difference is that the loop status will be FAIL in this case.

[**Nesting WHILE loops**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-345)

WHILE loops can be nested and also combined with other control structures:

**\*\*\* Test Cases \*\*\***

**Nesting WHILE**

${x} = Set Variable 10

WHILE ${x} > 0

${y} = Set Variable ${x}

WHILE ${y} > 0

${y} = Evaluate ${y} - 1

END

IF ${x} > 5

${x} = Evaluate ${x} - 1

ELSE

${x} = Evaluate ${x} - 2

END

END

[**Removing unnecessary keywords from outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-346)

WHILE loops with multiple iterations often create lots of output and considerably increase the size of the generated [output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) and [log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) files. It is possible to [remove or flatten unnecessary keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) using --removekeywords and --flattenkeywords command line options.

[**2.9.3   Loop control using BREAK and CONTINUE**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-347)

Both [FOR](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for) and [WHILE](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#while) loop execution can be controlled with BREAK and CONTINUE statements. The former exits the whole loop prematurely and the latter stops executing the current loop iteration and continues to the next one. In practice they have the same semantics as break and continue statements in Python, Java, and many other programming languages.

Both BREAK and CONTINUE are typically used conditionally with [IF/ELSE](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else) or [TRY/EXCEPT](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#try-except) structures, and especially the [inline IF](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-if) syntax is often convenient with them. These statements must be used in the loop body, possibly inside the aforementioned control structures, and using them in keyword called in the loop body is invalid.

**\*\*\* Test Cases \*\*\***

**BREAK with FOR**

${text} = Set Variable zero

FOR ${var} IN one two three

IF '${var}' == 'two' BREAK

${text} = Set Variable ${text}-${var}

END

Should Be Equal ${text} zero-one

**CONTINUE with FOR**

${text} = Set Variable zero

FOR ${var} IN one two three

IF '${var}' == 'two' CONTINUE

${text} = Set Variable ${text}-${var}

END

Should Be Equal ${text} zero-one-three

**CONTINUE and BREAK with WHILE**

WHILE True

TRY

${value} = Do Something

EXCEPT

CONTINUE

Do something with value ${value}

BREAK

END

**Invalid BREAK usage**

[**Documentation**] BREAK and CONTINUE can only be used in the loop body,

... not in keywords used in the loop.

FOR ${var} IN one two three

Invalid BREAK

END

**\*\*\* Keywords \*\*\***

**Invalid BREAK**

[**Documentation**] This keyword fails due to invalid syntax.

BREAK

**Note**

BREAK and CONTINUE statements are new in Robot Framework 5.0 similarly as WHILE. Earlier versions supported controlling FOR loops using [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Exit For Loop*, *Exit For Loop If*, *Continue For Loop* and *Continue For Loop If*. These keywords still continue to work, but they will be deprecated and removed in the future.

**Note**

Also the [RETURN](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return) statement can be used to a exit loop. It only works when loops are used inside a [user keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords).

[**2.9.4   IF/ELSE syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-348)

Sometimes there is a need to execute some keywords conditionally. Starting from Robot Framework 4.0 there is a separate IF/ELSE syntax, but there are also [other ways to execute keywords conditionally](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#other-ways-to-execute-keywords-conditionally). Notice that if the logic gets complicated, it is typically better to move it into a [test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries).

[**Basic IF syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-349)

Robot Framework's native IF syntax starts with IF (case-sensitive) and ends with END (case-sensitive). The IF marker requires exactly one value that is the condition to evaluate. Keywords to execute if the condition is true are on their own rows between the IF and END markers. Indenting keywords in the IF block is highly recommended but not mandatory.

In the following example keywords *Some keyword* and *Another keyword* are executed if ${rc} is greater than zero:

**\*\*\* Test Cases \*\*\***

**Example**

IF ${rc} > 0

Some keyword

Another keyword

END

The condition is evaluated in Python so that Python builtins like len() are available and modules are imported automatically to support usages like platform.system() == 'Linux' and math.ceil(${x}) == 1. Normal variables like ${rc} in the above example are replaced before evaluation, but variables are also available in the evaluation namespace using the special $rc syntax. The latter approach is handy when the string representation of the variable cannot be used in the condition directly. For example, strings require quoting and multiline strings and string themselves containing quotes cause additional problems. For more information and examples related the evaluation syntax see the [Evaluating expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#evaluating-expressions) appendix.

[**ELSE branches**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-350)

Like most other languages supporting conditional execution, Robot Framework IF syntax also supports ELSE branches that are executed if the IF condition is not true.

In this example *Some keyword* is executed if ${rc} is greater than zero and *Another keyword* is executed otherwise:

**\*\*\* Test Cases \*\*\***

**Example**

IF ${rc} > 0

Some keyword

ELSE

Another keyword

END

[**ELSE IF branches**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-351)

Robot Framework also supports ELSE IF branches that have their own condition that is evaluated if the initial condition is not true. There can be any number of ELSE IF branches and they are gone through in the order they are specified. If one of the ELSE IF conditions is true, the block following it is executed and remaining ELSE IF branches are ignored. An optional ELSE branch can follow ELSE IF branches and it is executed if all conditions are false.

In the following example different keyword is executed depending on is ${rc} positive, negative, zero, or something else like a string or None:

**\*\*\* Test Cases \*\*\***

**Example**

IF $rc > 0

Positive keyword

ELSE IF $rc < 0

Negative keyword

ELSE IF $rc == 0

Zero keyword

ELSE

Fail Unexpected rc: ${rc}

END

Notice that this example uses the ${rc} variable in the special $rc format to avoid evaluation failures if it is not a number. See the aforementioned [Evaluating expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#evaluating-expressions) appendix for more information about this syntax.

[**Inline IF**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-352)

Normal IF/ELSE structure is a bit verbose if there is a need to execute only a single statement. An alternative to it is using inline IF syntax where the statement to execute follows the IF marker and condition directly and no END marker is needed. For example, the following two keywords are equivalent:

**\*\*\* Keyword \*\*\***

**Normal IF**

IF $condition1

Keyword argument

END

IF $condition2

RETURN

END

**Inline IF**

IF $condition1 Keyword argument

IF $condition2 RETURN

The inline IF syntax supports also ELSE and ELSE IF branches:

**\*\*\* Keyword \*\*\***

**Inline IF/ELSE**

IF $condition Keyword argument ELSE Another Keyword

**Inline IF/ELSE IF/ELSE**

IF $cond1 Keyword 1 ELSE IF $cond2 Keyword 2 ELSE IF $cond3 Keyword 3 ELSE Keyword 4

As the latter example above demonstrates, inline IF with several ELSE IF and ELSE branches starts to get hard to understand. Long inline IF structures can be [split into multiple lines](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows) using the common ... continuation syntax, but using a normal IF/ELSE structure or moving the logic into a [test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) is probably a better idea. Each inline IF branch can contain only one statement. If more statements are needed, normal IF/ELSE structure needs to be used instead.

If there is a need for an assignment with inline IF, the variable or variables to assign must be before the starting IF. Otherwise the logic is exactly the same as when [assigning variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) based on keyword return values. If assignment is used and no branch is run, the variable gets value None.

**\*\*\* Keyword \*\*\***

**Inline IF/ELSE with assignment**

${var} = IF $condition Keyword argument ELSE Another Keyword

**Inline IF/ELSE with assignment having multiple variables**

${host} ${port} = IF $production Get Production Config ELSE Get Testing Config

**Note**

Inline IF syntax is new in Robot Framework 5.0.

[**Nested IF structures**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-353)

IF structures can be nested with each others and with [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops). This is illustrated by the following example using advanced features such as [FOR-IN-ENUMERATE loop](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-in-enumerate-loop), [named-only arguments with user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-user-keywords) and [inline Python evaluation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-python-evaluation) syntax (${{len(${items})}}):

**\*\*\* Keyword \*\*\***

**Log items**

[**Arguments**] @{items} ${log\_values}=True

IF not ${items}

Log to console No items.

ELSE IF len(${items}) == 1

IF ${log\_values}

Log to console One item: ${items}[0]

ELSE

Log to console One item.

END

ELSE

Log to console ${{len(${items})}} items.

IF ${log\_values}

FOR ${index} ${item} IN ENUMERATE @{items} start=1

Log to console Item ${index}: ${item}

END

END

END

**\*\*\* Test Cases \*\*\***

**No items**

Log items

**One item without logging value**

Log items xxx log\_values=False

**Multiple items**

Log items a b c

[**Other ways to execute keywords conditionally**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-354)

There are also other methods to execute keywords conditionally:

* The name of the keyword used as a setup or a teardown with [tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown), [suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) or [keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-teardown) can be specified using a variable. This facilitates changing them, for example, from the command line.
* The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword* takes a keyword to actually execute as an argument and it can thus be a variable. The value of the variable can, for example, be got dynamically from an earlier keyword or given from the command line.
* The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Run Keyword If* and *Run Keyword Unless* execute a named keyword only if a certain expression is true or false, respectively. The new IF/ELSE syntax explained above is generally recommended, though.
* Another [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword, *Set Variable If*, can be used to set variables dynamically based on a given expression.
* There are several [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords that allow executing a named keyword only if a test case or test suite has failed or passed.

[**2.9.5   TRY/EXCEPT syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-355)

When a keyword fails, Robot Framework's default behavior is to stop the current test and executes its possible [teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). There can, however, be needs to handle these failures during execution as well. Robot Framework 5.0 introduces native TRY/EXCEPT syntax for this purpose, but there also [other ways to handle errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#other-ways-to-handle-errors).

Robot Framework's TRY/EXCEPT syntax is inspired by Python's [exception handling](https://docs.python.org/tutorial/errors.html#handling-exceptions) syntax. It has same TRY, EXCEPT, ELSE and FINALLY branches as Python and they also mostly work the same way. A difference is that Python uses lower case try, except, etc. but with Robot Framework all this kind of syntax must use upper case letters. A bigger difference is that with Python exceptions are objects and with Robot Framework you are dealing with error messages as strings.

[**Catching exceptions with EXCEPT**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-356)

The basic TRY/EXCEPT syntax can be used to handle failures based on error messages:

**\*\*\* Test Cases \*\*\***

**First example**

TRY

Some Keyword

EXCEPT Error message

Error Handler Keyword

END

Keyword Outside

In the above example, if Some Keyword passes, the EXCEPT branch is not run and execution continues after the TRY/EXCEPT structure. If the keyword fails with a message Error message (case-sensitive), the EXCEPT branch is executed. If the EXCEPT branch succeeds, execution continues after the TRY/EXCEPT structure. If it fails, the test fails and remaining keywords are not executed. If Some Keyword fails with any other exception, that failure is not handled and the test fails without executing remaining keywords.

There can be more than one EXCEPT branch. In that case they are matched one by one and the first matching branch is executed. One EXCEPT can also have multiple messages to match, and such a branch is executed if any of its messages match. In all these cases messages can be specified using variables in addition to literal strings.

**\*\*\* Test Cases \*\*\***

**Multiple EXCEPT branches**

TRY

Some Keyword

EXCEPT Error message *# Try matching this first.*

Error Handler 1

EXCEPT Another error *# Try this if above did not match.*

Error Handler 2

EXCEPT ${message} *# Last match attempt, this time using a variable.*

Error Handler 3

END

**Multiple messages with one EXCEPT**

TRY

Some Keyword

EXCEPT Error message Another error ${message} *# Match any of these.*

Error handler

END

It is also possible to have an EXCEPT without messages, in which case it matches any error. There can be only one such EXCEPT and it must follow possible other EXCEPT branches:

**\*\*\* Test Cases \*\*\***

**Match any error**

TRY

Some Keyword

EXCEPT *# Match any error.*

Error Handler

END

**Match any after testing more specific errors**

TRY

Some Keyword

EXCEPT Error message *# Try matching this first*

Error Handler 1

EXCEPT *# Match any that did not match the above.*

Error Handler 2

END

[**Matching errors using patterns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-357)

By default matching an error using EXCEPT requires an exact match. That can be changed using a configuration option type= as an argument to the except clause. Valid values for the option are GLOB, REGEXP or START (case-insensitive) to make the match a [glob pattern match](https://en.wikipedia.org/wiki/Glob_(programming)), a [regular expression match](https://en.wikipedia.org/wiki/Regular_expression), or to match only the beginning of the error, respectively. Using value LITERAL has the same effect as the default behavior. If an EXCEPT has multiple messages, this option applies to all of them. The value of the option can be defined with a variable as well.

**\*\*\* Settings \*\*\***

${regexp} regexp

**\*\*\* Test Cases \*\*\***

**Glob pattern**

TRY

Some Keyword

EXCEPT ValueError: \* type=GLOB

Error Handler 1

EXCEPT [Ee]rror ?? occurred ${pattern} type=glob

Error Handler 2

END

**Regular expression**

TRY

Some Keyword

EXCEPT ValueError: .\* type=${regexp}

Error Handler 1

EXCEPT [Ee]rror \\d+ occurred type=Regexp *# Backslash needs to be escaped.*

Error Handler 2

END

**Match start**

TRY

Some Keyword

EXCEPT ValueError: ${beginning} type=start

Error Handler

END

**Explicit exact match**

TRY

Some Keyword

EXCEPT ValueError: invalid literal for int() with base 10: 'ooops' type=LITERAL

Error Handler

EXCEPT Error 13 occurred type=LITERAL

Error Handler 2

END

**Note**

Remember that the backslash character often used with regular expressions is an [escape character](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) in Robot Framework data. It thus needs to be escaped with another backslash when using it in regular expressions.

[**Capturing error message**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-358)

When [matching errors using patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#matching-errors-using-patterns) and when using EXCEPT without any messages to match any error, it is often useful to know the actual error that occurred. Robot Framework supports that by making it possible to capture the error message into a variable by adding AS  ${var} at the end of the EXCEPT statement:

**\*\*\* Test Cases \*\*\***

**Capture error**

TRY

Some Keyword

EXCEPT GLOB: ValueError: \* AS ${error}

Error Handler 1 ${error}

EXCEPT REGEXP: [Ee]rror \\d+ GLOB: ${pattern} AS ${error}

Error Handler 2 ${error}

EXCEPT AS ${error}

Error Handler 3 ${error}

END

[**Using ELSE to execute keywords when there are no errors**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-359)

Optional ELSE branches make it possible to execute keywords if there is no error. There can be only one ELSE branch and it is allowed only after one or more EXCEPT branches:

**\*\*\* Test Cases \*\*\***

**ELSE branch**

TRY

Some Keyword

EXCEPT X

Log Error 'X' occurred!

EXCEPT Y

Log Error 'Y' occurred!

ELSE

Log No error occurred!

END

Keyword Outside

In the above example, if Some Keyword passes, the ELSE branch is executed, and if it fails with message X or Y, the appropriate EXCEPT branch run. In all these cases execution continues after the whole TRY/EXCEPT/ELSE structure. If Some Keyword fail any other way, EXCEPT and ELSE branches are not run and the TRY/EXCEPT/ELSE structure fails.

To handle both the case when there is any error and when there is no error, it is possible to use an EXCEPT without any message in combination with an ELSE:

**\*\*\* Test Cases \*\*\***

**Handle everything**

TRY

Some Keyword

EXCEPT AS ${err}

Log Error occurred: ${err}

ELSE

Log No error occurred!

END

[**Using FINALLY to execute keywords regardless are there errors or not**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-360)

Optional FINALLY branches make it possible to execute keywords both when there is an error and when there is not. They are thus suitable for cleaning up after a keyword execution somewhat similarly as [teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). There can be only one FINALLY branch and it must always be last. They can be used in combination with EXCEPT and ELSE branches and having also TRY/FINALLY structure is possible:

**\*\*\* Test Cases \*\*\***

**TRY/EXCEPT/ELSE/FINALLY**

TRY

Some keyword

EXCEPT

Log Error occurred!

ELSE

Log No error occurred.

FINALLY

Log Always executed.

END

**TRY/FINALLY**

Open Connection

TRY

Use Connection

FINALLY

Close Connection

END

[**Other ways to handle errors**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-361)

There are also other methods to execute keywords conditionally:

* The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword And Expect Error* executes a named keyword and expects that it fails with a specified error message. It is basically the same as using TRY/EXCEPT with a specified message. The syntax to specify the error message is also identical except that this keyword uses glob pattern matching, not exact match, by default. Using the native TRY/EXCEPT functionality is generally recommended unless there is a need to support older Robot Framework versions that do not support it.
* The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword And Ignore Error* executes a named keyword and returns its status as string PASS or FAIL along with possible return value or error message. It is basically the same as using TRY/EXCEPT/ELSE so that EXCEPT catches all errors. Using the native syntax is recommended unless old Robot Framework versions need to be supported.
* The [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword And Return Status* executes a named keyword and returns its status as a Boolean true or false. It is a wrapper for the aforementioned *Run Keyword And Ignore Error*. The native syntax is nowadays recommended instead.
* [Test teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) and [keyword teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-teardown) can be used for cleaning up activities similarly as FINALLY branches.
* When keywords are implemented in Python based [libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries), all Python's error handling features are readily available. This is the recommended approach especially if needed logic gets more complicated.

[**2.10   Advanced features**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-214)

* [2.10.1   Handling keywords with same names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-keywords-with-same-names)
  + [Keyword scopes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-scopes)
  + [Specifying a keyword explicitly](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-a-keyword-explicitly)
  + [Specifying explicit priority between libraries and resources](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-explicit-priority-between-libraries-and-resources)
* [2.10.2   Timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timeouts)
  + [Test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout)
  + [User keyword timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-timeout)
* [2.10.3   Parallel execution of keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#parallel-execution-of-keywords)

[**2.10.1   Handling keywords with same names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-362)

Keywords that are used with Robot Framework are either [library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) or [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords). The former come from [standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#standard-libraries) or [external libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#external-libraries), and the latter are either created in the same file where they are used or then imported from [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files). When many keywords are in use, it is quite common that some of them have the same name, and this section describes how to handle possible conflicts in these situations.

[**Keyword scopes**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-363)

When only a keyword name is used and there are several keywords with that name, Robot Framework attempts to determine which keyword has the highest priority based on its scope. The keyword's scope is determined on the basis of how the keyword in question is created:

1. Created as a user keyword in the currently executed [test case file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files). These keywords have the highest priority and they are always used, even if there are other keywords with the same name elsewhere.
2. Created in a resource file and imported either directly or indirectly from another resource file. This is the second-highest priority.
3. Created in an external test library. These keywords are used, if there are no user keywords with the same name. However, if there is a keyword with the same name in the standard library, a warning is displayed.
4. Created in a standard library. These keywords have the lowest priority.

[**Specifying a keyword explicitly**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-364)

Scopes alone are not a sufficient solution, because there can be keywords with the same name in several libraries or resources, and thus, they provide a mechanism to use only the keyword of the highest priority. In such cases, it is possible to use *the full name of the keyword*, where the keyword name is prefixed with the name of the resource or library and a dot is a delimiter.

With library keywords, the long format means only using the format *LibraryName.Keyword Name*. For example, the keyword *Run* from the [OperatingSystem](https://robotframework.org/robotframework/latest/libraries/OperatingSystem.html) library could be used as *OperatingSystem.Run*, even if there was another *Run* keyword somewhere else. If the library is in a module or package, the full module or package name must be used (for example, *com.company.Library.Some Keyword*). If a custom name is given to a library using the [WITH NAME syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name-to-test-library), the specified name must be used also in the full keyword name.

Resource files are specified in the full keyword name, similarly as library names. The name of the resource is derived from the basename of the resource file without the file extension. For example, the keyword *Example* in a resource file *myresources.html* can be used as *myresources.Example*. Note that this syntax does not work, if several resource files have the same basename. In such cases, either the files or the keywords must be renamed. The full name of the keyword is case-, space- and underscore-insensitive, similarly as normal keyword names.

[**Specifying explicit priority between libraries and resources**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-365)

If there are multiple conflicts between keywords, specifying all the keywords in the long format can be quite a lot work. Using the long format also makes it impossible to create dynamic test cases or user keywords that work differently depending on which libraries or resources are available. A solution to both of these problems is specifying the keyword priorities explicitly using the keyword *Set Library Search Order* from the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library.

**Note**

Although the keyword has the word *library* in its name, it works also with resource files. As discussed above, keywords in resources always have higher priority than keywords in libraries, though.

The *Set Library Search Order* accepts an ordered list or libraries and resources as arguments. When a keyword name in the test data matches multiple keywords, the first library or resource containing the keyword is selected and that keyword implementation used. If the keyword is not found from any of the specified libraries or resources, execution fails for conflict the same way as when the search order is not set.

For more information and examples, see the documentation of the keyword.

[**2.10.2   Timeouts**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-366)

Sometimes keywords may take exceptionally long time to execute or just hang endlessly. Robot Framework allows you to set timeouts both for [test cases](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestCase) and [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords), and if a test or keyword is not finished within the specified time, the keyword that is currently being executed is forcefully stopped.

Stopping keywords in this manner may leave the library, the test environment or the system under test to an unstable state, and timeouts are recommended only when there is no safer option available. In general, libraries should be implemented so that keywords cannot hang or that they have their own timeout mechanism.

[**Test case timeout**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-367)

The test case timeout can be set either by using the *Test Timeout* setting in the Setting section or the *[Timeout]* setting with individual test cases. *Test Timeout* defines a default timeout for all the test cases in that suite, whereas *[Timeout]* applies a timeout to a particular test case and overrides the possible default value.

Using an empty *[Timeout]* means that the test has no timeout even when *Test Timeout* is used. It is also possible to use explicit NONE value for this purpose. The timeout is effectively ignored also if its value is zero or negative.

Regardless of where the test timeout is defined, the value given to it contains the duration of the timeout. The duration must be given in Robot Framework's [time format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format), that is, either directly in seconds like 10 or in a format like 1 minute 30 seconds. Timeouts can also be specified as [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) making it possible to give them, for example, from the command line.

If there is a timeout and it expires, the keyword that is currently running is stopped and the test case fails. Keywords executed as part of [test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown) are not interrupted if a test timeout occurs, though, but the test is nevertheless marked failed. If a keyword in teardown may hang, it can be stopped by using [user keyword timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-timeout).

**\*\*\* Settings \*\*\***

**Test Timeout** 2 minutes

**\*\*\* Test Cases \*\*\***

**Default timeout**

[**Documentation**] Default timeout from Settings is used.

Some Keyword argument

**Override**

[**Documentation**] Override default, use 10 seconds timeout.

[**Timeout**] 10

Some Keyword argument

**Variables**

[**Documentation**] It is possible to use variables too.

[**Timeout**] ${TIMEOUT}

Some Keyword argument

**No timeout**

[**Documentation**] Empty timeout means no timeout even when Test Timeout has been used.

[**Timeout**]

Some Keyword argument

**No timeout 2**

[**Documentation**] Disabling timeout with NONE works too and is more explicit.

[**Timeout**] NONE

Some Keyword argument

[**User keyword timeout**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-368)

Timeouts can be set for user keywords using the *[Timeout]* setting. The syntax is exactly the same as with [test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout), but user keyword timeouts do not have any default value. If a user keyword timeout is specified using a variable, the value can be given also as a keyword argument.

**\*\*\* Keywords \*\*\***

**Hardcoded**

[**Arguments**] ${arg}

[**Timeout**] 1 minute 42 seconds

Some Keyword ${arg}

**Configurable**

[**Arguments**] ${arg} ${timeout}

[**Timeout**] ${timeout}

Some Keyword ${arg}

**Run Keyword with Timeout**

[**Arguments**] ${keyword} @{args} &{kwargs} ${timeout}=1 minute

[**Documentation**] Wrapper that runs another keyword with a configurable timeout.

[**Timeout**] ${timeout}

Run Keyword ${keyword} @{args} &{kwargs}

A user keyword timeout is applicable during the execution of that user keyword. If the total time of the whole keyword is longer than the timeout value, the currently executed keyword is stopped. User keyword timeouts are applicable also during a test case teardown, whereas test timeouts are not.

If both the test case and some of its keywords (or several nested keywords) have a timeout, the active timeout is the one with the least time left.

**Note**

With earlier Robot Framework versions it was possible to specify a custom error message to use if a timeout expires. This functionality was deprecated in Robot Framework 3.0.1 and removed in Robot Framework 3.2.

[**2.10.3   Parallel execution of keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-369)

When parallel execution is needed, it must be implemented in test library level so that the library executes the code on background. Typically this means that the library needs a keyword like *Start Something* that starts the execution and returns immediately, and another keyword like *Get Results From Something* that waits until the result is available and returns it. See [Process](https://robotframework.org/robotframework/latest/libraries/Process.html) library keywords *Start Process* and *Wait For Process* for an example.

[**3   Executing test cases**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-73)

* [3.1   Basic usage](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#basic-usage-1)
* [3.2   Test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-execution)
* [3.3   Task execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-execution)
* [3.4   Post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs)
* [3.5   Configuring execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-execution)
* [3.6   Created outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#created-outputs)

[**3.1   Basic usage**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-370)

Robot Framework test cases are executed from the command line, and the end result is, by default, an [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) in XML format and an HTML [report](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) and [log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file). After the execution, output files can be combined and otherwise [post-processed](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs) with the Rebot tool.

* [3.1.1   Starting test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-test-execution)
  + [Synopsis](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#synopsis)
  + [Specifying test data to be executed](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-test-data-to-be-executed)
* [3.1.2   Using command line options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-command-line-options)
  + [Using options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-options)
  + [Short and long options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#short-and-long-options)
  + [Setting option values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-option-values)
  + [Disabling options accepting no values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#disabling-options-accepting-no-values)
  + [Simple patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns)
  + [Tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns)
  + [ROBOT\_OPTIONS and REBOT\_OPTIONS environment variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#robot-options-and-rebot-options-environment-variables)
* [3.1.3   Test results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-results)
  + [Command line output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#command-line-output)
  + [Generated output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generated-output-files)
  + [Return codes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes)
  + [Errors and warnings during execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution)
* [3.1.4   Argument files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files)
  + [Argument file syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-file-syntax)
  + [Using argument files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-argument-files)
  + [Reading argument files from standard input](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reading-argument-files-from-standard-input)
* [3.1.5   Getting help and version information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-help-and-version-information)
* [3.1.6   Creating start-up scripts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-start-up-scripts)
  + [Shell script example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#shell-script-example)
  + [Batch file example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#batch-file-example)
  + [Python example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#python-example)
* [3.1.7   Making *\*.robot* files executable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#making-robot-files-executable)
* [3.1.8   Debugging problems](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debugging-problems)
  + [Using the Python debugger (pdb)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-the-python-debugger-pdb)

[**3.1.1   Starting test execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-376)

[**Synopsis**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-377)

robot [options] data

python -m robot [options] data

python path/to/robot/ [options] data

Execution is normally started using the robot command created as part of [installation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#installation-instructions). Alternatively it is possible to execute the installed robot module using the selected Python interpreter. This is especially convenient if Robot Framework has been installed under multiple Python versions. Finally, if you know where the installed robot directory exists, it can be executed using Python as well.

Regardless of execution approach, the path (or paths) to the test data to be executed is given as an argument after the command. Additionally, different command line options can be used to alter the test execution or generated outputs in many ways.

[**Specifying test data to be executed**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-378)

Robot Framework test cases are created in [files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files) and [directories](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories), and they are executed by giving the path to the file or directory in question to the selected runner script. The path can be absolute or, more commonly, relative to the directory where tests are executed from. The given file or directory creates the top-level test suite, which gets its name, unless overridden with the --name [option](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-name), from the [file or directory name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation). Different execution possibilities are illustrated in the examples below. Note that in these examples, as well as in other examples in this section, only the robot script is used, but other execution approaches could be used similarly.

robot tests.robot

robot path/to/my\_tests/

robot c:\robot\tests.robot

**Note**

When executing a [directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories), all files and directories starting with a dot (*.*) or an underscore (*\_*) are ignored and, by default, only files with the *.robot* extension executed. See the [Selecting files to parse](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse) section for more details.

It is also possible to give paths to several test case files or directories at once, separated with spaces. In this case, Robot Framework creates the top-level test suite automatically, and the specified files and directories become its child test suites. The name of the created test suite is got from child suite names by catenating them together with an ampersand (&) and spaces. For example, the name of the top-level suite in the first example below is *My Tests & Your Tests*. These automatically created names are often quite long and complicated. In most cases, it is thus better to use the --name option for overriding it, as in the second example below:

robot my\_tests.robot your\_tests.robot

robot --name Example path/to/tests/pattern\_\*.robot

[**3.1.2   Using command line options**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-379)

Robot Framework provides a number of command line options that can be used to control how test cases are executed and what outputs are generated. This section explains the option syntax, and what options actually exist. How they can be used is discussed elsewhere in this chapter.

[**Using options**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-380)

When options are used, they must always be given between the runner script and the data sources. For example:

robot -L debug my\_tests.robot

robot --include smoke --variable HOST:10.0.0.42 path/to/tests/

[**Short and long options**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-381)

Options always have a long name, such as --name, and the most frequently needed options also have a short name, such as -N. In addition to that, long options can be shortened as long as they are unique. For example, --logle DEBUG works, while --lo log.html does not, because the former matches only --loglevel, but the latter matches several options. Short and shortened options are practical when executing test cases manually, but long options are recommended in [start-up scripts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#start-up-scripts), because they are easier to understand.

The long option format is case-insensitive, which facilitates writing option names in an easy-to-read format. For example, --SuiteStatLevel is equivalent to, but easier to read than --suitestatlevel.

[**Setting option values**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-382)

Most of the options require a value, which is given after the option name. Both short and long options accept the value separated from the option name with a space, as in --include tag or -i tag. With long options, the separator can also be the equals sign, for example --include=tag, and with short options the separator can be omitted, as in -itag.

Some options can be specified several times. For example, --variable VAR1:value --variable VAR2:another sets two variables. If the options that take only one value are used several times, the value given last is effective.

[**Disabling options accepting no values**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-383)

Options accepting no values can be disabled by using the same option again with no prefix added or dropped. The last option has precedence regardless of how many times options are used. For example, --dryrun --dryrun --nodryrun --nostatusrc --statusrc would not activate the dry-run mode and would return normal status rc.

[**Simple patterns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-384)

Many command line options take arguments as *simple patterns*. These [glob-like patterns](http://en.wikipedia.org/wiki/Glob_(programming)) are matched according to the following rules:

* \* matches any string, even an empty string.
* ? matches any single character.
* [abc] matches one character in the bracket.
* [!abc] 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.
* Unless noted otherwise, pattern matching is case, space, and underscore insensitive.

Examples:

--test Example\* # Matches tests with name starting 'Example'.

--test Example[1-2] # Matches tests 'Example1' and 'Example2'.

--include f?? # Matches tests with a tag that starts with 'f' is three characters long.

All matching in above examples is case, space and underscore insensitive. For example, the second example would also match test named example 1.

If the matched text happens to contain some of the wildcard characters and they need to be matched literally, it is possible to do that by using the [...] syntax. The pattern [\*] matches the literal \* character, [?] matches ?, and [[] matches [. Lone [ and ] do not need to be escaped.

**Note**

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

[**Tag patterns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-385)

Most tag related options accept arguments as *tag patterns*. They support same wildcards as [simple patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns) (e.g. examp??, ex\*le), but they also support AND, OR and NOT operators explained below. These operators can be used for combining two or more individual tags or patterns together.

AND or &

The whole pattern matches if all individual patterns match. AND and & are equivalent:

--include fooANDbar # Matches tests containing tags 'foo' and 'bar'.

--exclude xx&yy&zz # Matches tests containing tags 'xx', 'yy', and 'zz'.

OR

The whole pattern matches if any individual pattern matches:

--include fooORbar # Matches tests containing either tag 'foo' or tag 'bar'.

--exclude xxORyyORzz # Matches tests containing any of tags 'xx', 'yy', or 'zz'.

NOT

The whole pattern matches if the pattern on the left side matches but the one on the right side does not. If used multiple times, none of the patterns after the first NOT must not match:

--include fooNOTbar # Matches tests containing tag 'foo' but not tag 'bar'.

--exclude xxNOTyyNOTzz # Matches tests containing tag 'xx' but not tag 'yy' or tag 'zz'.

The pattern can also start with NOT in which case the pattern matches if the pattern after NOT does not match:

--include NOTfoo # Matches tests not containing tag 'foo'

--include NOTfooANDbar # Matches tests not containing tags 'foo' and 'bar'

The above operators can also be used together. The operator precedence, from highest to lowest, is AND, OR and NOT:

--include xANDyORz # Matches tests containing either tags 'x' and 'y', or tag 'z'.

--include xORyNOTz # Matches tests containing either tag 'x' or 'y', but not tag 'z'.

--include xNOTyANDz # Matches tests containing tag 'x', but not tags 'y' and 'z'.

Although tag matching itself is case-insensitive, all operators are case-sensitive and must be written with upper case letters. If tags themselves happen to contain upper case AND, OR or NOT, they need to specified using lower case letters to avoid accidental operator usage:

--include port # Matches tests containing tag 'port', case-insensitively

--include PORT # Matches tests containing tag 'P' or 'T', case-insensitively

--exclude handoverORportNOTnotification

[**ROBOT\_OPTIONS and REBOT\_OPTIONS environment variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-386)

Environment variables ROBOT\_OPTIONS and REBOT\_OPTIONS can be used to specify default options for [test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-test-execution) and [result post-processing](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs), respectively. The options and their values must be defined as a space separated list and they are placed in front of any explicit options on the command line. The main use case for these environment variables is setting global default values for certain options to avoid the need to repeat them every time tests are run or Rebot used.

export ROBOT\_OPTIONS="--outputdir results --tagdoc 'mytag:Example doc with spaces'"

robot tests.robot

export REBOT\_OPTIONS="--reportbackground green:yellow:red"

rebot --name example output.xml

[**3.1.3   Test results**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-387)

[**Command line output**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-388)

The most visible output from test execution is the output displayed in the command line. All executed test suites and test cases, as well as their statuses, are shown there in real time. The example below shows the output from executing a simple test suite with only two test cases:

==============================================================================

Example test suite

==============================================================================

First test :: Possible test documentation | PASS |

------------------------------------------------------------------------------

Second test | FAIL |

Error message is displayed here

==============================================================================

Example test suite | FAIL |

2 tests, 1 passed, 1 failed

==============================================================================

Output: /path/to/output.xml

Report: /path/to/report.html

Log: /path/to/log.html

There is also a notification on the console whenever a top-level keyword in a test case ends. A green dot is used if a keyword passes and a red F if it fails. These markers are written to the end of line and they are overwritten by the test status when the test itself ends. Writing the markers is disabled if console output is redirected to a file.

[**Generated output files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-389)

The command line output is very limited, and separate output files are normally needed for investigating the test results. As the example above shows, three output files are generated by default. The first one is in XML format and contains all the information about test execution. The second is a higher-level report and the third is a more detailed log file. These files and other possible output files are discussed in more detail in the section [Different output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-output-files).

[**Return codes**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-390)

Runner scripts communicate the overall test execution status to the system running them using return codes. When the execution starts successfully and no tests fail, the return code is zero. All possible return codes are explained in the table below.

| *Possible return codes* | |
| --- | --- |
| **RC** | **Explanation** |
| 0 | All tests passed. |
| 1-249 | Returned number of tests failed. |
| 250 | 250 or more failures. |
| 251 | Help or version information printed. |
| 252 | Invalid test data or command line options. |
| 253 | Test execution stopped by user. |
| 255 | Unexpected internal error. |

Return codes should always be easily available after the execution, which makes it easy to automatically determine the overall execution status. For example, in bash shell the return code is in special variable $?, and in Windows it is in %ERRORLEVEL% variable. If you use some external tool for running tests, consult its documentation for how to get the return code.

The return code can be set to 0 even if there are failures using the --NoStatusRC command line option. This might be useful, for example, in continuous integration servers where post-processing of results is needed before the overall status of test execution can be determined.

**Note**

Same return codes are also used with [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot).

[**Errors and warnings during execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-391)

During the test execution there can be unexpected problems like failing to import a library or a resource file or a keyword being [deprecated](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#deprecating-keywords). Depending on the severity such problems are categorized as errors or warnings and they are written into the console (using the standard error stream), shown on a separate *Test Execution Errors* section in log files, and also written into Robot Framework's own [system log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log). Normally these errors and warnings are generated by Robot Framework itself, but libraries can also log [errors and warnings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings). Example below illustrates how errors and warnings look like in the log file.

|  |  |  |
| --- | --- | --- |
| 20090322 19:58:42.528 | ERROR | Error in file '/home/robot/tests.robot' in table 'Setting' in element on row 2: Resource file 'resource.robot' does not exist |
| 20090322 19:58:43.931 | WARN | Keyword 'SomeLibrary.Example Keyword' is deprecated. Use keyword `Other Keyword` instead. |

[**3.1.4   Argument files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-392)

Argument files allow placing all or some command line options and arguments into an external file where they will be read. This avoids the problems with characters that are problematic on the command line. If lot of options or arguments are needed, argument files also prevent the command that is used on the command line growing too long.

Argument files are taken into use with --argumentfile (-A) option along with possible other command line options.

**Note**

Unlike other [long command line options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#short-and-long-options), --argumentfile cannot be given in shortened format like --argumentf.

[**Argument file syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-393)

Argument files can contain both command line options and paths to the test data, one option or data source per line. Both short and long options are supported, but the latter are recommended because they are easier to understand. Argument files can contain any characters without escaping, but spaces in the beginning and end of lines are ignored. Additionally, empty lines and lines starting with a hash mark (#) are ignored:

--doc This is an example (where "special characters" are ok!)

--metadata X:Value with spaces

--variable VAR:Hello, world!

# This is a comment

path/to/my/tests

In the above example the separator between options and their values is a single space. It is possible to use either an equal sign (=) or any number of spaces. As an example, the following three lines are identical:

--name An Example

--name=An Example

--name An Example

If argument files contain non-ASCII characters, they must be saved using UTF-8 encoding.

[**Using argument files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-394)

Argument files can be used either alone so that they contain all the options and paths to the test data, or along with other options and paths. When an argument file is used with other arguments, its contents are placed into the original list of arguments to the same place where the argument file option was. This means that options in argument files can override options before it, and its options can be overridden by options after it. It is possible to use --argumentfile option multiple times or even recursively:

robot --argumentfile all\_arguments.robot

robot --name Example --argumentfile other\_options\_and\_paths.robot

robot --argumentfile default\_options.txt --name Example my\_tests.robot

robot -A first.txt -A second.txt -A third.txt tests.robot

[**Reading argument files from standard input**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-395)

Special argument file name STDIN can be used to read arguments from the standard input stream instead of a file. This can be useful when generating arguments with a script:

generate\_arguments.sh | robot --argumentfile STDIN

generate\_arguments.sh | robot --name Example --argumentfile STDIN tests.robot

[**3.1.5   Getting help and version information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-396)

Both when executing test cases and when post-processing outputs, it is possible to get command line help with the option --help (-h). These help texts have a short general overview and briefly explain the available command line options.

All runner scripts also support getting the version information with the option --version. This information also contains Python version and the platform type:

$ robot --version

Robot Framework 5.0 (Python 3.8.12 on darwin)

C:\>rebot --version

Rebot 5.0 (Python 3.7.0 on win32)

[**3.1.6   Creating start-up scripts**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-397)

Test cases are often executed automatically by a continuous integration system or some other mechanism. In such cases, there is a need to have a script for starting the test execution, and possibly also for post-processing outputs somehow. Similar scripts are also useful when running tests manually, especially if a large number of command line options are needed or setting up the test environment is complicated.

In UNIX-like environments, shell scripts provide a simple but powerful mechanism for creating custom start-up scripts. Windows batch files can also be used, but they are more limited and often also more complicated. A platform-independent alternative is using Python or some other high-level programming language. Regardless of the language, it is recommended that long option names are used, because they are easier to understand than the short names.

[**Shell script example**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-398)

In this example, the same web tests in the login directory are executed with different browsers and the results combined afterwards using [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot). The script also accepts command line options itself and simply forwards them to the robot command using the handy $\* variable:

*#!/bin/bash*

robot --name Firefox --variable BROWSER:Firefox --output out/fx.xml --log none --report none $\* login

robot --name IE --variable BROWSER:IE --output out/ie.xml --log none --report none $\* login

rebot --name Login --outputdir out --output login.xml out/fx.xml out/ie.xml

[**Batch file example**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-399)

Implementing the above shell script example using batch files is not very complicated either. Notice that arguments to batch files can be forwarded to executed commands using %\*:

@**echo** off

robot --name Firefox --variable BROWSER:Firefox --output out\fx.xml --log none --report none %\* login

robot --name IE --variable BROWSER:IE --log none --output out\ie.xml --report none %\* login

rebot --name Login --outputdir out --output login.xml out\fx.xml out\ie.xml

**Note**

Prior to Robot Framework 3.1 robot and rebot commands were implemented as batch files on Windows and using them in another batch file required prefixing the whole command with call.

[**Python example**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-400)

When start-up scripts gets more complicated, implementing them using shell scripts or batch files is not that convenient. This is especially true if both variants are needed and same logic needs to be implemented twice. In such situations it is often better to switch to Python. It is possible to execute Robot Framework from Python using the [subprocess module](https://docs.python.org/library/subprocess.html), but often using Robot Framework's own [programmatic API](https://robot-framework.readthedocs.io/) is more convenient. The easiest APIs to use are robot.run\_cli and robot.rebot\_cli that accept same command line arguments than the robot and rebot commands.

The following example implements the same logic as the earlier shell script and batch file examples. In Python arguments to the script itself are available in sys.argv:

*#!/usr/bin/env python*

**import** **sys**

**from** **robot** **import** run\_cli, rebot\_cli

common = ['--log', 'none', '--report', 'none'] + sys.argv[1:] + ['login']

run\_cli(['--name', 'Firefox', '--variable', 'BROWSER:Firefox', '--output', 'out/fx.xml'] + common, exit=**False**)

run\_cli(['--name', 'IE', '--variable', 'BROWSER:IE', '--output', 'out/ie.xml'] + common, exit=**False**)

rebot\_cli(['--name', 'Login', '--outputdir', 'out', 'out/fx.xml', 'out/ie.xml'])

**Note**

exit=False is needed because by default run\_cli exits to system with the correct [return code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes). rebot\_cli does that too, but in the above example that is fine.

[**3.1.7   Making *\*.robot* files executable**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-401)

On UNIX-like operating systems it is possible to make *\*.robot* files executable by giving them execution permission and adding a [shebang](https://en.wikipedia.org/wiki/Shebang_(Unix)) like in this example:

*#!/usr/bin/env robot*

**\*\*\* Test Cases \*\*\***

**Example**

Log to console Executing!

If the above content would be in a file *example.robot* and that file would be executable, it could be executed from the command line like below. Starting from Robot Framework 3.2, individually executed files can have any extension, or no extension at all, so the same would work also if the file would be named just *example*.

./example.robot

This trick does not work when executing a directory but can be handy when executing a single file. It is probably more often useful when [automating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks) than when automating tests.

[**3.1.8   Debugging problems**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-402)

A test case can fail because the system under test does not work correctly, in which case the test has found a bug, or because the test itself is buggy. The error message explaining the failure is shown on the [command line output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#command-line-output) and in the [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file), and sometimes the error message alone is enough to pinpoint the problem. More often that not, however, [log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) are needed because they have also other log messages and they show which keyword actually failed.

When a failure is caused by the tested application, the error message and log messages ought to be enough to understand what caused it. If that is not the case, the test library does not provide [enough information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#communicating-with-robot-framework) and needs to be enhanced. In this situation running the same test manually, if possible, may also reveal more information about the issue.

Failures caused by test cases themselves or by keywords they use can sometimes be hard to debug. If the error message, for example, tells that a keyword is used with wrong number of arguments fixing the problem is obviously easy, but if a keyword is missing or fails in unexpected way finding the root cause can be harder. The first place to look for more information is the [execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) section in the log file. For example, an error about a failed test library import may well explain why a test has failed due to a missing keyword.

If the log file does not provide enough information by default, it is possible to execute tests with a lower [log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). For example tracebacks showing where in the code the failure occurred are logged using the DEBUG level, and this information is invaluable when the problem is in an individual library keyword.

Logged tracebacks do not contain information about methods inside Robot Framework itself. If you suspect an error is caused by a bug in the framework, you can enable showing internal traces by setting environment variable ROBOT\_INTERNAL\_TRACES to any non-empty value.

If the log file still does not have enough information, it is a good idea to enable the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) and see what information it provides. It is also possible to add some keywords to the test cases to see what is going on. Especially [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Log* and *Log Variables* are useful. If nothing else works, it is always possible to search help from [mailing lists](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#mailing-lists) or elsewhere.

[**Using the Python debugger (pdb)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-403)

It is also possible to use the [pdb](http://docs.python.org/library/pdb.html) module from the Python standard library to set a break point and interactively debug a running test. The typical way of invoking pdb by inserting:

**import** **pdb**; pdb.set\_trace()

at the location you want to break into debugger will not work correctly with Robot Framework, as the standard output stream is redirected during keyword execution. Instead, you can use the following:

**import** **sys**, **pdb**; pdb.Pdb(stdout=sys.\_\_stdout\_\_).set\_trace()

from within a python library or alternatively:

*Evaluate* *pdb.Pdb(stdout=sys.\_\_stdout\_\_).set\_trace()* *modules=sys, pdb*

can be used directly in a test case.

[**3.2   Test execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-371)

This section describes how the test suite structure created from the parsed test data is executed, how test status is determined, and how to continue executing a test case if there are failures, and how to stop the whole test execution gracefully.

* [3.2.1   Execution flow](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#execution-flow)
  + [Executed suites and tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executed-suites-and-tests)
  + [Setups and teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setups-and-teardowns)
  + [Execution order](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#execution-order)
* [3.2.2   Test and suite statuses](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-and-suite-statuses)
  + [PASS](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pass)
  + [FAIL](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#fail)
  + [SKIP](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skip)
  + [Migrating from criticality to SKIP](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#migrating-from-criticality-to-skip)
  + [Suite status](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-status)
* [3.2.3   Continue on failure](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure)
  + [*Run Keyword And Ignore Error* and *Run Keyword And Expect Error* keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#run-keyword-and-ignore-error-and-run-keyword-and-expect-error-keywords)
  + [*Run Keyword And Warn On Failure*](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#run-keyword-and-warn-on-failure)
  + [Special failures from keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#special-failures-from-keywords)
  + [*Run Keyword And Continue On Failure* keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#run-keyword-and-continue-on-failure-keyword)
  + [Controlling continue on failure using reserved tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#controlling-continue-on-failure-using-reserved-tags)
  + [Execution continues on teardowns automatically](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#execution-continues-on-teardowns-automatically)
  + [All top-level keywords are executed when tests have templates](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#all-top-level-keywords-are-executed-when-tests-have-templates)
* [3.2.4   Stopping test execution gracefully](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution-gracefully)
  + [Pressing Ctrl-C](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pressing-ctrl-c)
  + [Using signals](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-signals)
  + [Using keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-keywords)
  + [Stopping when first test case fails](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-when-first-test-case-fails)
  + [Stopping on parsing or execution error](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-on-parsing-or-execution-error)
  + [Handling teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-teardowns)

[**3.2.1   Execution flow**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-404)

[**Executed suites and tests**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-405)

Test cases are always executed within a test suite. A test suite created from a [test case file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-files) has tests directly, whereas suites created from [directories](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories) have child test suites which either have tests or their own child suites. By default all the tests in an executed suite are run, but it is possible to [select tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases) using options --test, --suite, --include and --exclude. Suites containing no tests are ignored.

The execution starts from the top-level test suite. If the suite has tests they are executed one-by-one, and if it has suites they are executed recursively in depth-first order. When an individual test case is executed, the keywords it contains are run in a sequence. Normally the execution of the current test ends if any of the keywords fails, but it is also possible to [continue after failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure). The exact [execution order](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#execution-order) and how possible [setups and teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setups-and-teardowns) affect the execution are discussed in the following sections.

[**Setups and teardowns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-406)

Setups and teardowns can be used on [test suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown), [test case](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) and [user keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown) levels.

**Suite setup**

If a test suite has a setup, it is executed before its tests and child suites. If the suite setup passes, test execution continues normally. If it fails, all the test cases the suite and its child suites contain are marked failed. The tests and possible suite setups and teardowns in the child test suites are not executed.

Suite setups are often used for setting up the test environment. Because tests are not run if the suite setup fails, it is easy to use suite setups for verifying that the environment is in state in which the tests can be executed.

**Suite teardown**

If a test suite has a teardown, it is executed after all its test cases and child suites. Suite teardowns are executed regardless of the test status and even if the matching suite setup fails. If the suite teardown fails, all tests in the suite are marked failed afterwards in reports and logs.

Suite teardowns are mostly used for cleaning up the test environment after the execution. To ensure that all these tasks are done, [all the keywords used in the teardown are executed](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure) even if some of them fail.

**Test setup**

Possible test setup is executed before the keywords of the test case. If the setup fails, the keywords are not executed. The main use for test setups is setting up the environment for that particular test case.

**Test teardown**

Possible test teardown is executed after the test case has been executed. It is executed regardless of the test status and also if test setup has failed.

Similarly as suite teardown, test teardowns are used mainly for cleanup activities. Also they are executed fully even if some of their keywords fail.

**Keyword teardown**

[User keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) cannot have setups, but they can have teardowns that work exactly like other teardowns. Keyword teardowns are run after the keyword is executed otherwise, regardless the status, and they are executed fully even if some of their keywords fail.

[**Execution order**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-407)

Test cases in a test suite are executed in the same order as they are defined in the test case file. Test suites inside a higher level test suite are executed in case-insensitive alphabetical order based on the file or directory name. If multiple files and/or directories are given from the command line, they are executed in the order they are given.

If there is a need to use certain test suite execution order inside a directory, it is possible to add prefixes like *01* and *02* into file and directory names. Such prefixes are not included in the generated test suite name if they are separated from the base name of the suite with two underscores:

01\_\_my\_suite.robot -> My Suite

02\_\_another\_suite.robot -> Another Suite

If the alphabetical ordering of test suites inside suites is problematic, a good workaround is giving them separately in the required order. This easily leads to overly long start-up commands, but [argument files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files) allow listing files nicely one file per line.

It is also possible to [randomize the execution order](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#randomizing-execution-order) using the --randomize option.

[**3.2.2   Test and suite statuses**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-408)

This section explains how tests can get [PASS](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pass), [FAIL](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#fail) or [SKIP](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skip) status and how the [suite status](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-status) is determined based on test statuses.

**Note**

The SKIP status is new in Robot Framework 4.0.

[**PASS**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-409)

A test gets the PASS status if it is executed and none of the keywords it contains fails.

**Prematurely passing tests**

Normally all keywords are executed, but it is also possible to use [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Pass Execution* and *Pass Execution If* to stop execution with the PASS status and not run the remaining keywords.

How *Pass Execution* and *Pass Execution If* behave in different situations is explained below:

* When used in any [setup or teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setups-and-teardowns) (suite, test or keyword), these keywords pass that setup or teardown. Possible teardowns of the started keywords are executed. Test execution or statuses are not affected otherwise.
* When used in a test case outside setup or teardown, the keywords pass that particular test case. Possible test and keyword teardowns are executed.
* Possible [continuable failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure) that occur before these keyword are used, as well as failures in teardowns executed afterwards, will fail the execution.
* It is mandatory to give an explanation message why execution was interrupted, and it is also possible to modify test case tags. For more details, and usage examples, see the [documentation of these keywords](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html).

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 [skip](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skip) the test.

[**FAIL**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-410)

The most common reason for a test to get the FAIL status is that one of the keywords it contains fails. The keyword itself can fail by [raising an exception](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reporting-keyword-status) or the keyword can be called incorrectly. Other reasons for failures include syntax errors and the test being empty.

If a [suite setup](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) fails, tests in that suite are marked failed without running them. If a [suite teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) fails, tests are marked failed retroactively.

[**SKIP**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-411)

Starting from Robot Framework 4.0, tests can get also SKIP status in addition to PASS and FAIL. There are many different ways to get this status.

**Skipping before execution**

The command line option --skip can be used to skip specified tests without running them at all. It works based on [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases) and supports [tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns) like examp?? and tagANDanother. If it is used multiple times, all tests matching any of specified tags or tag patterns are skipped:

--skip require-network

--skip windowsANDversion9?

--skip python2.\* --skip python3.[0-6]

Starting from RF 5.0, a test case can also be skipped by tagging the test with the reserved tag robot:skip:

**\*\*\* Test Cases \*\*\***

**Example**

[**Tags**] robot:skip

Log This is not executed

The difference between --skip and --exclude is that with the latter tests are [omitted from the execution altogether](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) and they will not be shown in logs and reports. With the former they are included, but not actually executed, and they will be visible in logs and reports.

**Skipping dynamically during execution**

Tests can get the skip status during execution in various ways:

* Using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Skip* anywhere in the test case, including setup or teardown. Using *Skip* keyword has two effects: the test gets the SKIP status and rest of the test is not executed. However, if the test has a teardown, it will be run.
* Using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Skip If* which takes a condition and skips the test if the condition is true.
* [Library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) may also trigger skip behavior by using a special exceptions. This is explained the [Skipping tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipping-tests) section in the [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) chapter.
* If [suite setup](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) is skipped using any of the above means, all tests in the suite are skipped without executing them.
* If [suite teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown) is skipped, all tests will be marked skipped retroactively.

**Automatically skipping failed tests**

The command line option --skiponfailure can be used to automatically mark failed tests skipped. It works based on [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases) and supports [tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns) like the --skip option discussed above:

--skiponfailure not-ready

--skiponfailure experimentalANDmobile

Starting from RF 5.0, the reserved tag robot:skip-on-failure can alternatively be used to achieve the same effect as above:

**\*\*\* Test Cases \*\*\***

**Example**

[**Tags**] robot:skip-on-failure

Fail this test will be marked as skipped instead of failed

The motivation for this functionality is allowing execution of tests that are not yet ready or that are testing a functionality that is not yet ready. Instead of such tests failing, they will be marked skipped and their tags can be used to separate them from possible other skipped tests.

[**Migrating from criticality to SKIP**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-412)

Earlier Robot Framework versions supported criticality concept that allowed marking tests critical or non-critical. By default all tests were critical, but the --critical and --noncritical options could be used to configure that. The difference between critical and non-critical tests was that non-critical tests were not included when determining the final status for an executed test suite or for the whole test run. In practice the test status was two dimensional having PASS and FAIL in one axis and criticality on the other.

Non-critical failed tests were in many ways similar to the current skipped tests. Because these features are similar and having both SKIP and criticality would have created strange test statuses like non-critical SKIP, the criticality concept was removed in Robot Framework 4.0 when the SKIP status was introduced. The problems with criticality are explained in more detail in the [issue that proposed removing it](https://github.com/robotframework/robotframework/issues/3624).

The main use case for the criticality concept was being able to run tests that are not yet ready or that are testing a functionality that is not yet ready. This use case is nowadays covered by the skip-on-failure functionality discussed in the previous section.

To ease migrating from criticality to skipping, the old --noncritical option worked as an alias for the new --skiponfailure in Robot Framework 4.0 and also the old --critical option was preserved. Both old options were deprecated and they were removed in Robot Framework 5.0.

[**Suite status**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-413)

Suite status is determined solely based on statuses of the tests it contains:

* If any test has failed, suite status is FAIL.
* If there are no failures but at least one test has passed, suite status is PASS.
* If all tests have been skipped or the are no tests at all, suite status is SKIP.

[**3.2.3   Continue on failure**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-414)

Normally test cases are stopped immediately when any of their keywords fail. This behavior shortens test execution time and prevents subsequent keywords hanging or otherwise causing problems if the system under test is in unstable state. This has the drawback that often subsequent keywords would give more information about the state of the system. Hence Robot Framework offers several features to continue after failures.

[***Run Keyword And Ignore Error* and *Run Keyword And Expect Error* keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-415)

[BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Run Keyword And Ignore Error* and *Run Keyword And Expect Error* handle failures so that test execution is not terminated immediately. Though, using these keywords for this purpose often adds extra complexity to test cases, so the following features are worth considering to make continuing after failures easier.

[***Run Keyword And Warn On Failure***](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-416)

[BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword And Warn On Failure* handles failure similar to *Run Keyword And Ignore Error* in the sense that test execution is not terminated immediately, but will report failures as a warning message.

[**Special failures from keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-417)

[Library keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) report failures using exceptions, and it is possible to use special exceptions to tell the core framework that execution can continue regardless the failure. How these exceptions can be created is explained in the [Continuable failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continuable-failures) section in the [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) section.

When a test ends and there has been one or more continuable failure, the test will be marked failed. If there are more than one failure, all of them will be enumerated in the final error message:

Several failures occurred:

1) First error message.

2) Second error message ...

Test execution ends also if a normal failure occurs after continuable failures. Also in that case all the failures will be listed in the final error message.

The return value from failed keywords, possibly assigned to a variable, is always the Python None.

[***Run Keyword And Continue On Failure* keyword**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-418)

[BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Run Keyword And Continue On Failure* allows converting any failure into a continuable failure. These failures are handled by the framework exactly the same way as continuable failures originating from library keywords.

[**Controlling continue on failure using reserved tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-419)

All keywords executed as part of test cases or user keywords which are tagged with the reserved tag robot:continue-on-failure are considered continuable by default.

Thus, the following two test cases *Test 1* and *Test 2* behave identically:

**\*\*\* Test Cases \*\*\***

**Test 1**

Run Keyword and Continue on Failure Should be Equal 1 2

User Keyword 1

**Test 2**

[**Tags**] robot:continue-on-failure

Should be Equal 1 2

User Keyword 2

**\*\*\* Keywords \*\*\***

**User Keyword 1**

Run Keyword and Continue on Failure Should be Equal 3 4

Log this message is logged

**User Keyword 2**

[**Tags**] robot:continue-on-failure

Should be Equal 3 4

Log this message is logged

These tags also influence continue-on-failure in FOR loops and within IF/ELSE branches. The below test case will execute the test 10 times, no matter if the "Perform some test keyword" failed or not.

**\*\*\* Test Cases \*\*\***

**Test Case**

[**Tags**] robot:continue-on-failure

FOR ${index} IN RANGE 10

Perform some test

END

Setting robot:continue-on-failure within a test case will not propagate the continue on failure behaviour into user keywords executed from within this test case (same is true for user keywords executed from within a user keyword with the reserved tag set).

To support use cases where the behaviour should propagate from test cases into user keywords (and/or from user keywords into other user keywords), the reserved tag robot:recursive-continue-on-failure can be used. The below examples executes all the keywords listed.

**\*\*\* Test Cases \*\*\***

**Test**

[**Tags**] robot:recursive-continue-on-failure

Should be Equal 1 2

User Keyword 1

Log log from test case

**\*\*\* Keywords \*\*\***

**User Keyword 1**

Should be Equal 3 4

Log log from keyword 1

User Keyword 2

**User Keyword 2**

Should be Equal 5 6

Log log from keyword 2

The robot:continue-on-failure and robot:recursive-continue-on-failure tags are new in Robot Framework 4.1.

[**Execution continues on teardowns automatically**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-420)

To make it sure that all the cleanup activities are taken care of, the continue on failure mode is automatically on in [test and suite teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setups-and-teardowns). In practice this means that in teardowns all the keywords in all levels are always executed.

[**All top-level keywords are executed when tests have templates**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-421)

When using [test templates](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates), all the data rows are always executed to make it sure that all the different combinations are tested. In this usage continuing is limited to the top-level keywords, and inside them the execution ends normally if there are non-continuable failures.

[**3.2.4   Stopping test execution gracefully**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-422)

Sometimes there is a need to stop the test execution before all the tests have finished, but so that logs and reports are created. Different ways how to accomplish this are explained below. In all these cases the remaining test cases are marked failed.

The tests that are automatically failed get robot:exit tag and the generated report will include NOT robot:exit [combined tag pattern](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generating-combined-tag-statistics) to easily see those tests that were not skipped. Note that the test in which the exit happened does not get the robot:exit tag.

**Note**

Prior to Robot Framework 3.1, the special tag was named robot-exit.

[**Pressing Ctrl-C**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-423)

The execution is stopped when Ctrl-C is pressed in the console where the tests are running. The execution is stopped immediately, but reports and logs are still generated.

If Ctrl-C is pressed again, the execution ends immediately and reports and logs are not created.

[**Using signals**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-424)

On UNIX-like machines it is possible to terminate test execution using signals INT and TERM. These signals can be sent from the command line using kill command, and sending signals can also be easily automated.

[**Using keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-425)

The execution can be stopped also by the executed keywords. There is a separate *Fatal Error* [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword for this purpose, and custom keywords can use [fatal exceptions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution) when they fail.

[**Stopping when first test case fails**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-426)

If option --exitonfailure (-X) is used, test execution stops immediately if any test fails. The remaining tests are marked as failed without actually executing them.

[**Stopping on parsing or execution error**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-427)

Robot Framework separates *failures* caused by failing keywords from *errors* caused by, for example, invalid settings or failed test library imports. By default these errors are reported as [test execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution), but errors themselves do not fail tests or affect execution otherwise. If --exitonerror option is used, however, all such errors are considered fatal and execution stopped so that remaining tests are marked failed. With parsing errors encountered before execution even starts, this means that no tests are actually run.

[**Handling teardowns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-428)

By default teardowns of the tests and suites that have been started are executed even if the test execution is stopped using one of the methods above. This allows clean-up activities to be run regardless how execution ends.

It is also possible to skip teardowns when execution is stopped by using --skipteardownonexit option. This can be useful if, for example, clean-up tasks take a lot of time.

[**3.3   Task execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-372)

Robot Framework can be used also for other automation purposes than test automation, and starting from Robot Framework 3.1 it is possible to explicitly [create](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks) and execute tasks. For most parts task execution and test execution work the same way, and this section explains the differences.

* [3.3.1   Generic automation mode](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generic-automation-mode)
* [3.3.2   Task related command line options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-related-command-line-options)

[**3.3.1   Generic automation mode**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-429)

When Robot Framework is used execute a file and it notices that the file has tasks, not tests, it automatically sets itself into the generic automation mode. This mode does not change the actual execution at all, but when logs and reports are created, they use term *task*, not *test*. They have, for example, headers like Task Log and Task Statistics instead of Test Log and Test Statistics.

The generic automation mode can also be enabled by using the --rpa option. In that case the executed files can have either tests or tasks. Alternatively --norpa can be used to force the test automation mode even if executed files contain tasks. If neither of these options are used, it is an error to execute multiple files so that some have tests and others have tasks.

The execution mode is stored in the generated [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) and read by [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot) if outputs are post-processed. The mode can also [be set when using Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#controlling-execution-mode) if necessary.

[**3.3.2   Task related command line options**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-430)

All normal command line options can be used when executing tasks. If there is a need to [select only certain tasks for execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases), --task can be used instead of --test. Additionally the aforementioned --rpa can be used to control the execution mode.

[**3.4   Post-processing outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-373)

[XML output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) that are generated during the test execution can be post-processed afterwards by the Rebot tool, which is an integral part of Robot Framework. It is used automatically when test reports and logs are generated during the test execution, and using it separately allows creating custom reports and logs as well as combining and merging results.

* [3.4.1   Using Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-rebot)
  + [Synopsis](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#synopsis-1)
  + [Specifying options and arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-options-and-arguments)
  + [Return codes with Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes-with-rebot)
  + [Controlling execution mode](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#controlling-execution-mode)
* [3.4.2   Creating different reports and logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-different-reports-and-logs)
* [3.4.3   Combining outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#combining-outputs)
* [3.4.4   Merging outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#merging-outputs)
  + [Merging re-executed tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#merging-re-executed-tests)
  + [Merging suites executed in pieces](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#merging-suites-executed-in-pieces)

[**3.4.1   Using Rebot**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-431)

[**Synopsis**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-432)

rebot [options] outputs

python -m robot.rebot [options] outputs

python path/to/robot/rebot.py [options] outputs

The most common way to use Rebot is using the rebot command. Alternatively it is possible to execute the installed robot.rebot module or the robot/rebot.py file using the selected Python interpreter.

[**Specifying options and arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-433)

The basic syntax for using Rebot is exactly the same as when [starting test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-test-execution) and also most of the command line options are identical. The main difference is that arguments to Rebot are [XML output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) instead of test data files or directories.

[**Return codes with Rebot**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-434)

Return codes from Rebot are exactly same as when [running tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes).

[**Controlling execution mode**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-435)

Rebot notices have [tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-test-execution) or [tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-execution) been run, and by default preserves the execution mode. The mode affects logs and reports so that in the former case they will use term *test* like Test Log and Test Statistics, and in the latter case term *task* like Task Log and Task Statistics.

Rebot also supports using --rpa or --norpa options to set the execution mode explicitly. This is necessary if multiple output files are processed and they have conflicting modes.

[**3.4.2   Creating different reports and logs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-436)

You can use Rebot for creating the same reports and logs that are created automatically during the test execution. Of course, it is not sensible to create the exactly same files, but, for example, having one report with all test cases and another with only some subset of tests can be useful:

rebot output.xml

rebot path/to/output\_file.xml

rebot --include smoke --name Smoke\_Tests c:\results\output.xml

Another common usage is creating only the output file when running tests (log and report generation can be disabled with --log NONE --report NONE) and generating logs and reports later. Tests can, for example, be executed on different environments, output files collected to a central place, and reports and logs created there.

[**3.4.3   Combining outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-437)

An important feature in Rebot is its ability to combine outputs from different test execution rounds. This capability allows, for example, running the same test cases on different environments and generating an overall report from all outputs. Combining outputs is extremely easy, all that needs to be done is giving several output files as arguments:

rebot output1.xml output2.xml

rebot outputs/\*.xml

When outputs are combined, a new top-level test suite is created so that test suites in the given output files are its child suites. This works the same way when [multiple test data files or directories are executed](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-test-data-to-be-executed), and also in this case the name of the top-level test suite is created by joining child suite names with an ampersand (&) and spaces. These automatically generated names are not that good, and it is often a good idea to use --name to give a more meaningful name:

rebot --name Browser\_Compatibility firefox.xml opera.xml safari.xml ie.xml

rebot --include smoke --name Smoke\_Tests c:\results\\*.xml

[**3.4.4   Merging outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-438)

If same tests are re-executed or a single test suite executed in pieces, combining results like discussed above creates an unnecessary top-level test suite. In these cases it is typically better to merge results instead. Merging is done by using --merge (-R) option which changes the way how Rebot combines two or more output files. This option itself takes no arguments and all other command line options can be used with it normally:

rebot --merge --name Example original.xml merged.xml

How merging works in practice is explained in the following sections discussing its two main use cases.

[**Merging re-executed tests**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-439)

There is often a need to re-execute a subset of tests, for example, after fixing a bug in the system under test or in the tests themselves. This can be accomplished by [selecting test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases) by names (--test and --suite options), tags (--include and --exclude), or by previous status (--rerunfailed or --rerunfailedsuites).

Combining re-execution results with the original results using the default [combining outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#combining-outputs) approach does not work too well. The main problem is that you get separate test suites and possibly already fixed failures are also shown. In this situation it is better to use --merge (-R) option to tell Rebot to merge the results instead. In practice this means that tests from the latter test runs replace tests in the original. An exception to this rule is that [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped) tests in latter runs are ignored and original tests preserved.

This usage is best illustrated by a practical example using --rerunfailed and --merge together:

robot --output original.xml tests # first execute all tests

robot --rerunfailed original.xml --output rerun.xml tests # then re-execute failing

rebot --merge original.xml rerun.xml # finally merge results

The message of the merged tests contains a note that results have been replaced. The message also shows the old status and message of the test.

Merged results must always have same top-level test suite. Tests and suites in merged outputs that are not found from the original output are added into the resulting output. How this works in practice is discussed in the next section.

**Note**

Ignoring skipped tests in latter runs is new in Robot Framework 4.1.

[**Merging suites executed in pieces**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-440)

Another important use case for the --merge option is merging results got when running a test suite in pieces using, for example, --include and --exclude options:

robot --include smoke --output smoke.xml tests # first run some tests

robot --exclude smoke --output others.xml tests # then run others

rebot --merge smoke.xml others.xml # finally merge results

When merging outputs like this, the resulting output contains all tests and suites found from all given output files. If some test is found from multiple outputs, latest results replace the earlier ones like explained in the previous section. Also this merging strategy requires the top-level test suites to be same in all outputs.

[**3.5   Configuring execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-374)

This section explains different command line options that can be used for configuring the [test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-test-execution) or [post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs). Options related to generated output files are discussed in the [next section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#created-outputs).

* [3.5.1   Selecting files to parse](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse)
* [3.5.2   Selecting test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases)
  + [By test suite and test case names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-test-suite-and-test-case-names)
  + [By tag names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names)
  + [Re-executing failed test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#re-executing-failed-test-cases)
  + [Re-executing failed test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#re-executing-failed-test-suites)
  + [When no tests match selection](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#when-no-tests-match-selection)
* [3.5.3   Setting metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-metadata)
  + [Setting the name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-name)
  + [Setting the documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-documentation)
  + [Setting free metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-free-metadata)
  + [Setting tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-tags)
* [3.5.4   Configuring where to search libraries and other extensions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-where-to-search-libraries-and-other-extensions)
  + [Locations automatically in module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#locations-automatically-in-module-search-path)
  + [PYTHONPATH](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pythonpath)
  + [Using --pythonpath option](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-pythonpath-option)
  + [Configuring sys.path programmatically](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-sys-path-programmatically)
* [3.5.5   Setting variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables)
* [3.5.6   Dry run](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dry-run)
* [3.5.7   Randomizing execution order](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#randomizing-execution-order)
* [3.5.8   Programmatic modification of test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-test-data)
  + [Example: Select every Xth test](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#example-select-every-xth-test)
  + [Example: Exclude tests by name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#example-exclude-tests-by-name)
  + [Example: Disable setups and teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#example-disable-setups-and-teardowns)
* [3.5.9   Controlling console output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#controlling-console-output)
  + [Console output type](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-output-type)
  + [Console width](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-width)
  + [Console colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-colors)
  + [Console markers](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-markers)
* [3.5.10   Setting listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-listeners)

[**3.5.1   Selecting files to parse**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-441)

When executing a single file, Robot Framework tries to parse and run it regardless the name or the file extension. The file is expected to use the [plain text format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats) or, if it has *.rst* or *.rest* extension, the [reStructuredText format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "restructuredtext-format):

robot example.robot # Common case.

robot example.tsv # Must be compatible with the plain text format.

robot example.rst # reStructuredText format.

When executing a [directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-directories), Robot Framework ignores all files and directories starting with a dot (*.*) or an underscore (*\_*) and, by default, only parses files with the *.robot* extension. If files use other extensions, the --extension (-F) option must be used to explicitly tell the framework to parse also them. If there is a need to parse more than one kind of files, it is possible to use a colon : to separate extensions. Matching extensions is case insensitive and the leading . can be omitted:

robot path/to/tests/ # Parse only \*.robot files.

robot --extension TSV path/to/tests # Parse only \*.tsv files.

robot -F robot:rst path/to/tests # Parse \*.robot and \*.rst files.

If files in one format use different extensions like *.rst* and *.rest*, they must be specified separately. Using just one of them would mean that other files in that format are skipped.

**Note**

Prior to Robot Framework 3.1 also TXT, TSV and HTML files were parsed by default. Starting from Robot Framework 3.2 HTML files are not supported at all.

[**3.5.2   Selecting test cases**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-442)

Robot Framework offers several command line options for selecting which test cases to execute. The same options work also when [executing tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-tasks) and when post-processing outputs with [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot).

[**By test suite and test case names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-443)

Test suites and test cases can be selected by their names with the command line options --suite (-s) and --test (-t), respectively. Both of these options can be used several times to select several test suites or cases. Arguments to these options are case- and space-insensitive, and there can also be [simple patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns) matching multiple names. If both the --suite and --test options are used, only test cases in matching suites with matching names are selected.

--test Example

--test mytest --test yourtest

--test example\*

--test mysuite.mytest

--test \*.suite.mytest

--suite example-??

--suite mysuite --test mytest --test your\*

Using the --suite option is more or less the same as executing only the appropriate test case file or directory. One major benefit is the possibility to select the suite based on its parent suite. The syntax for this is specifying both the parent and child suite names separated with a dot. In this case, the possible setup and teardown of the parent suite are executed.

--suite parent.child

--suite myhouse.myhousemusic --test jack\*

Selecting individual test cases with the --test option is very practical when creating test cases, but quite limited when running tests automatically. The --suite option can be useful in that case, but in general, selecting test cases by tag names is more flexible.

When [executing tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-tasks), it is possible to use the --task option as an alias for --test.

[**By tag names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-444)

It is possible to include and exclude test cases by [tag](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases) names with the --include (-i) and --exclude (-e) options, respectively. If the --include option is used, only test cases having a matching tag are selected, and with the --exclude option test cases having a matching tag are not. If both are used, only tests with a tag matching the former option, and not with a tag matching the latter, are selected:

--include example

--exclude not\_ready

--include regression --exclude long\_lasting

Both --include and --exclude can be used several times to match multiple tags. In that case a test is selected if it has a tag that matches any included tags, and also has no tag that matches any excluded tags.

In addition to specifying a tag to match fully, it is possible to use [tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns) where \* and ? are wildcards and AND, OR, and NOT operators can be used for combining individual tags or patterns together:

--include feature-4?

--exclude bug\*

--include fooANDbar

--exclude xxORyyORzz

--include fooNOTbar

Starting from RF 5.0, it is also possible to use the reserved tag robot:exclude to achieve the same effect as with using the --exclude option:

**\*\*\* Test Cases \*\*\***

**Example**

[**Tags**] robot:exclude

Fail This is not executed

Selecting test cases by tags is a very flexible mechanism and allows many interesting possibilities:

* A subset of tests to be executed before other tests, often called smoke tests, can be tagged with smoke and executed with --include smoke.
* Unfinished test can be committed to version control with a tag such as not\_ready and excluded from the test execution with --exclude not\_ready.
* Tests can be tagged with sprint-<num>, where <num> specifies the number of the current sprint, and after executing all test cases, a separate report containing only the tests for a certain sprint can be generated (for example, rebot --include sprint-42 output.xml).

[**Re-executing failed test cases**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-445)

Command line option --rerunfailed (-R) can be used to select all failed tests from an earlier [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) for re-execution. This option is useful, for example, if running all tests takes a lot of time and one wants to iteratively fix failing test cases.

robot tests # first execute all tests

robot --rerunfailed output.xml tests # then re-execute failing

Behind the scenes this option selects the failed tests as they would have been selected individually with the --test option. It is possible to further fine-tune the list of selected tests by using --test, --suite, --include and --exclude options.

Using an output not originating from executing the same tests that are run now causes undefined results. Additionally, it is an error if the output contains no failed tests. Using a special value NONE as the output is same as not specifying this option at all.

**Tip**

Re-execution results and original results can be [merged together](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#merging-outputs) using the --merge command line option.

[**Re-executing failed test suites**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-446)

Command line option --rerunfailedsuites (-S) can be used to select all failed suites from an earlier [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) for re-execution. Like --rerunfailed (-R), this option is useful when full test execution takes a lot of time. Note that all tests from a failed test suite will be re-executed, even passing ones. This option is useful when the tests in a test suite depends on each other.

Behind the scenes this option selects the failed suites as they would have been selected individually with the --suite option. It is possible to further fine-tune the list of selected tests by using --test, --suite, --include and --exclude options.

[**When no tests match selection**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-447)

By default when no tests match the selection criteria test execution fails with an error like:

[ ERROR ] Suite 'Example' with includes 'xxx' contains no test cases.

Because no outputs are generated, this behavior can be problematic if tests are executed and results processed automatically. Luckily a command line option --RunEmptySuite can be used to force the suite to be executed also in this case. As a result normal outputs are created but show zero executed tests. The same option can be used also to alter the behavior when an empty directory or a test case file containing no tests is executed.

Similar situation can occur also when processing output files with [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot). It is possible that no test match the used filtering criteria or that the output file contained no tests to begin with. By default executing Rebot fails in these cases, but it has a separate --ProcessEmptySuite option that can be used to alter the behavior. In practice this option works the same way as --RunEmptySuite when running tests.

[**3.5.3   Setting metadata**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-448)

[**Setting the name**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-449)

When Robot Framework parses test data, [test suite names are created from file and directory names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation). The name of the top-level test suite can, however, be overridden with the command line option --name (-N).

**Note**

Prior to Robot Framework 3.1, underscores in the value were converted to spaces. Nowadays values containing spaces need to be escaped or quoted like, for example, --name "My example".

[**Setting the documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-450)

In addition to [defining documentation in the test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation), documentation of the top-level suite can be given from the command line with the option --doc (-D). The value can contain simple [HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) and must be quoted if it contains spaces.

If the given documentation is a relative or absolute path pointing to an existing file, the actual documentation will be read from that file. This is especially convenient if the externally specified documentation is long or contains multiple lines.

Examples:

robot --doc "Example documentation" tests.robot

robot --doc doc.txt tests.robot # Documentation read from doc.txt if it exits.

**Note**

Reading documentation from an external file is new in Robot Framework 4.1.

Prior to Robot Framework 3.1, underscores in documentation were converted to spaces same way as with the --name option.

[**Setting free metadata**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-451)

[Free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) may also be given from the command line with the option --metadata (-M). The argument must be in the format name:value, where name the name of the metadata to set and value is its value. The value can contain simple [HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) and the whole argument must be quoted if it contains spaces. This option may be used several times to set multiple metadata values.

If the given value is a relative or absolute path pointing to an existing file, the actual value will be read from that file. This is especially convenient if the value is long or contains multiple lines. If the value should be a path to an existing file, not read from that file, the value must be separated with a space from the name: part.

Examples:

robot --metadata Name:Value tests.robot

robot --metadata "Another Name:Another value, now with spaces" tests.robot

robot --metadata "Read From File:meta.txt" tests.robot # Value read from meta.txt if it exists.

robot --metadata "Path As Value: meta.txt" tests.robot # Value always used as-is.

**Note**

Reading metadata value from an external file is new in Robot Framework 4.1.

Prior to Robot Framework 3.1, underscores in the value were converted to spaces same way as with the --name option.

[**Setting tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-452)

The command line option --settag (-G) can be used to set the given tag to all executed test cases. This option may be used several times to set multiple tags.

[**3.5.4   Configuring where to search libraries and other extensions**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-453)

When Robot Framework imports a [test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-library-to-import), [listener](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-listeners), or some other Python based extension, it uses the Python interpreter to import the module containing the extension from the system. The list of locations where modules are looked for is called *the module search path*, and its contents can be configured using different approaches explained in this section.

Robot Framework uses Python's module search path also when importing [resource and variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-and-variable-files) if the specified path does not match any file directly.

The module search path being set correctly so that libraries and other extensions are found is a requirement for successful test execution. If you need to customize it using approaches explained below, it is often a good idea to create a custom [start-up script](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#start-up-script).

[**Locations automatically in module search path**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-454)

Python interpreters have their own standard library as well as a directory where third party modules are installed automatically in the module search path. This means that test libraries [packaged using Python's own packaging system](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#packaging-libraries) are automatically installed so that they can be imported without any additional configuration.

[**PYTHONPATH**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-455)

Python reads additional locations to be added to the module search path from PYTHONPATH environment variables. If you want to specify more than one location in any of them, you need to separate the locations with a colon on UNIX-like machines (e.g. /opt/libs:$HOME/testlibs) and with a semicolon on Windows (e.g. D:\libs;%HOMEPATH%\testlibs).

Environment variables can be configured permanently system wide or so that they affect only a certain user. Alternatively they can be set temporarily before running a command, something that works extremely well in custom [start-up scripts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#start-up-scripts).

[**Using --pythonpath option**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-456)

Robot Framework has a separate command line option --pythonpath (-P) for adding locations to the module search path.

Multiple locations can be given by separating them with a colon (:) or a semicolon (;) or by using this option multiple times. If the value contains both colons and semicolons, it is split from semicolons. Paths can also be [glob patterns](https://en.wikipedia.org/wiki/Glob_(programming)) matching multiple paths, but they typically need to be escaped when used on the console.

Examples:

--pythonpath libs

--pythonpath /opt/testlibs:mylibs.zip:yourlibs

--pythonpath /opt/testlibs --pythonpath mylibs.zip --pythonpath yourlibs

--pythonpath c:\temp;d:\resources

--pythonpath lib/\\*.zip # '\*' is escaped

**Note**

Both colon and semicolon work regardless the operating system. Using semicolon is new in Robot Framework 5.0.

[**Configuring sys.path programmatically**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-457)

Python interpreters store the module search path they use as a list of strings in [sys.path](http://docs.python.org/library/sys.html" \l "sys.path) attribute. This list can be updated dynamically during execution, and changes are taken into account next time when something is imported.

[**3.5.5   Setting variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-458)

[Variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) can be set from the command line either [individually](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line) using the --variable (-v) option or through [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files) with the --variablefile (-V) option. Variables and variable files are explained in separate chapters, but the following examples illustrate how to use these options:

--variable name:value

--variable OS:Linux --variable IP:10.0.0.42

--variablefile path/to/variables.py

--variablefile myvars.py:possible:arguments:here

--variable ENVIRONMENT:Windows --variablefile c:\resources\windows.py

[**3.5.6   Dry run**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-459)

Robot Framework supports so called *dry run* mode where the tests are run normally otherwise, but the keywords coming from the test libraries are not executed at all. The dry run mode can be used to validate the test data; if the dry run passes, the data should be syntactically correct. This mode is triggered using option --dryrun.

The dry run execution may fail for following reasons:

* Using keywords that are not found.
* Using keywords with wrong number of arguments.
* Using user keywords that have invalid syntax.

In addition to these failures, normal [execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) are shown, for example, when test library or resource file imports cannot be resolved.

It is possible to disable dry run validation of specific [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-user-keywords) by adding a special robot:no-dry-run [keyword tag](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags) to them. This is useful if a keyword fails in the dry run mode for some reason, but work fine when executed normally.

**Note**

The dry run mode does not validate variables.

[**3.5.7   Randomizing execution order**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-460)

The test execution order can be randomized using option --randomize <what>[:<seed>], where <what> is one of the following:

tests

Test cases inside each test suite are executed in random order.

suites

All test suites are executed in a random order, but test cases inside suites are run in the order they are defined.

all

Both test cases and test suites are executed in a random order.

none

Neither execution order of test nor suites is randomized. This value can be used to override the earlier value set with --randomize.

It is possible to give a custom seed to initialize the random generator. This is useful if you want to re-run tests using the same order as earlier. The seed is given as part of the value for --randomize in format <what>:<seed> and it must be an integer. If no seed is given, it is generated randomly. The executed top level test suite automatically gets [metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) named *Randomized* that tells both what was randomized and what seed was used.

Examples:

robot --randomize tests my\_test.robot

robot --randomize all:12345 path/to/tests

[**3.5.8   Programmatic modification of test data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-461)

If the provided built-in features to modify test data before execution are not enough, Robot Framework makes it possible to do custom modifications programmatically. This is accomplished by creating a so called *pre-run modifier* and activating it using the --prerunmodifier option.

Pre-run modifiers should be implemented as visitors that can traverse through the executable test suite structure and modify it as needed. The visitor interface is explained as part of the [Robot Framework API documentation](http://robot-framework.readthedocs.org/en/master/autodoc/robot.model.html#module-robot.model.visitor), and it possible to modify executed [test suites](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite), [test cases](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestCase) and [keywords](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.Keyword) using it. The examples below ought to give an idea of how pre-run modifiers can be used and how powerful this functionality is.

When a pre-run modifier is taken into use on the command line using the --prerunmodifier option, it can be specified either as a name of the modifier class or a path to the modifier file. If the modifier is given as a class name, the module containing the class must be in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path), and if the module name is different than the class name, the given name must include both like module.ModifierClass. If the modifier is given as a path, the class name must be same as the file name. For most parts this works exactly like when [importing a test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-library-to-import).

If a modifier requires arguments, like the examples below do, they can be specified after the modifier name or path using either a colon (:) or a semicolon (;) as a separator. If both are used in the value, the one used first is considered to be the actual separator. Starting from Robot Framework 4.0, arguments also support the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) as well as [argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) based on [type hints](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations) and [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values) the same way as keywords do.

If more than one pre-run modifier is needed, they can be specified by using the --prerunmodifier option multiple times. If similar modifying is needed before creating logs and reports, [programmatic modification of results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-results) can be enabled using the --prerebotmodifier option.

Pre-run modifiers are executed before other configuration affecting the executed test suite and test cases. Most importantly, options related to [selecting test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-test-cases) are processed after modifiers, making it possible to use options like --include also with possible dynamically added tests.

**Tip**

Modifiers are taken into use from the command line exactly the same way as [listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface). See the [Taking listeners into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-listeners-into-use) section for more information and examples.

**Note**

Prior to Robot Framework 3.2 pre-run modifiers were executed after other configuration.

[**Example: Select every Xth test**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-462)

The first example shows how a pre-run modifier can remove tests from the executed test suite structure. In this example only every Xth tests is preserved, and the X is given from the command line along with an optional start index.

*"""Pre-run modifier that selects only every Xth test for execution.*

*Starts from the first test by default. Tests are selected per suite.*

*"""*

**from** **robot.api** **import** SuiteVisitor

**class** **SelectEveryXthTest**(SuiteVisitor):

**def** \_\_init\_\_(self, x: int, start: int = 0):

self.x = x

self.start = start

**def** start\_suite(self, suite):

*"""Modify suite's tests to contain only every Xth."""*

suite.tests = suite.tests[self.start::self.x]

**def** end\_suite(self, suite):

*"""Remove suites that are empty after removing tests."""*

suite.suites = [s **for** s **in** suite.suites **if** s.test\_count > 0]

**def** visit\_test(self, test):

*"""Avoid visiting tests and their keywords to save a little time."""*

**pass**

If the above pre-run modifier is in a file *SelectEveryXthTest.py* and the file is in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path), it could be used like this:

# Specify the modifier as a path. Run every second test.

robot --prerunmodifier path/to/SelectEveryXthTest.py:2 tests.robot

# Specify the modifier as a name. Run every third test, starting from the second.

robot --prerunmodifier SelectEveryXthTest:3:1 tests.robot

**Note**

Argument conversion based on type hints like x: int in the above example is new in Robot Framework 4.0 and requires Python 3.

[**Example: Exclude tests by name**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-463)

Also the second example removes tests, this time based on a given name pattern. In practice it works like a negative version of the built-in --test option.

*"""Pre-run modifier that excludes tests by their name.*

*Tests to exclude are specified by using a pattern that is both case and space*

*insensitive and supports '\*' (match anything) and '?' (match single character)*

*as wildcards.*

*"""*

**from** **robot.api** **import** SuiteVisitor

**from** **robot.utils** **import** Matcher

**class** **ExcludeTests**(SuiteVisitor):

**def** \_\_init\_\_(self, pattern):

self.matcher = Matcher(pattern)

**def** start\_suite(self, suite):

*"""Remove tests that match the given pattern."""*

suite.tests = [t **for** t **in** suite.tests **if** **not** self.\_is\_excluded(t)]

**def** \_is\_excluded(self, test):

**return** self.matcher.match(test.name) **or** self.matcher.match(test.longname)

**def** end\_suite(self, suite):

*"""Remove suites that are empty after removing tests."""*

suite.suites = [s **for** s **in** suite.suites **if** s.test\_count > 0]

**def** visit\_test(self, test):

*"""Avoid visiting tests and their keywords to save a little time."""*

**pass**

Assuming the above modifier is in a file named *ExcludeTests.py*, it could be used like this:

# Exclude test named 'Example'.

robot --prerunmodifier path/to/ExcludeTests.py:Example tests.robot

# Exclude all tests ending with 'something'.

robot --prerunmodifier path/to/ExcludeTests.py:\*something tests.robot

[**Example: Disable setups and teardowns**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-464)

Sometimes when debugging tests it can be useful to disable setups or teardowns. This can be accomplished by editing the test data, but pre-run modifiers make it easy to do that temporarily for a single run:

*"""Pre-run modifiers for disabling suite and test setups and teardowns."""*

**from** **robot.api** **import** SuiteVisitor

**class** **SuiteSetup**(SuiteVisitor):

**def** start\_suite(self, suite):

suite.setup = **None**

**class** **SuiteTeardown**(SuiteVisitor):

**def** start\_suite(self, suite):

suite.teardown = **None**

**class** **TestSetup**(SuiteVisitor):

**def** start\_test(self, test):

test.setup = **None**

**class** **TestTeardown**(SuiteVisitor):

**def** start\_test(self, test):

test.teardown = **None**

Assuming that the above modifiers are all in a file named *disable.py* and this file is in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path), setups and teardowns could be disabled, for example, as follows:

# Disable suite teardowns.

robot --prerunmodifier disable.SuiteTeardown tests.robot

# Disable both test setups and teardowns by using '--prerunmodifier' twice.

robot --prerunmodifier disable.TestSetup --prerunmodifier disable.TestTeardown tests.robot

**Note**

Prior to Robot Framework 4.0 setup and teardown were accessed via the intermediate keywords attribute and, for example, suite setup was disabled like suite.keywords.setup = None.

[**3.5.9   Controlling console output**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-465)

There are various command line options to control how test execution is reported on the console.

[**Console output type**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-466)

The overall console output type is set with the --console option. It supports the following case-insensitive values:

verbose

Every test suite and test case is reported individually. This is the default.

dotted

Only show . for passed test, F for failed tests, s for skipped tests and x for tests which are skipped because [test execution exit](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution-gracefully). Failed tests are listed separately after execution. This output type makes it easy to see are there any failures during execution even if there would be a lot of tests.

quiet

No output except for [errors and warnings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings).

none

No output whatsoever. Useful when creating a custom output using, for example, [listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface).

Separate convenience options --dotted (-.) and --quiet are shortcuts for --console dotted and --console quiet, respectively.

Examples:

robot --console quiet tests.robot

robot --dotted tests.robot

[**Console width**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-467)

The width of the test execution output in the console can be set using the option --consolewidth (-W). The default width is 78 characters.

**Tip**

On many UNIX-like machines you can use handy $COLUMNS environment variable like --consolewidth $COLUMNS.

[**Console colors**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-468)

The --consolecolors (-C) option is used to control whether colors should be used in the console output. Colors are implemented using [ANSI colors](http://en.wikipedia.org/wiki/ANSI_escape_code) except on Windows where, by default, Windows APIs are used instead.

This option supports the following case-insensitive values:

auto

Colors are enabled when outputs are written into the console, but not when they are redirected into a file or elsewhere. This is the default.

on

Colors are used also when outputs are redirected. Does not work on Windows.

ansi

Same as on but uses ANSI colors also on Windows. Useful, for example, when redirecting output to a program that understands ANSI colors.

off

Colors are disabled.

[**Console markers**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-469)

Special markers . (success) and F (failure) are shown on the console when using the [verbose output](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-output-type) and top level keywords in test cases end. The markers allow following the test execution in high level, and they are erased when test cases end.

It is possible to configure when markers are used with --consolemarkers (-K) option. It supports the following case-insensitive values:

auto

Markers are enabled when the standard output is written into the console, but not when it is redirected into a file or elsewhere. This is the default.

on

Markers are always used.

off

Markers are disabled.

[**3.5.10   Setting listeners**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-470)

[Listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface) can be used to monitor the test execution. When they are taken into use from the command line, they are specified using the --listener command line option. The value can either be a path to a listener or a listener name. See the [Listener interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface) section for more details about importing listeners and using them in general.

[**3.6   Created outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-375)

Several output files are created when tests are executed, and all of them are somehow related to test results. This section discusses what outputs are created, how to configure where they are created, and how to fine-tune their contents.

* [3.6.1   Different output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-output-files)
  + [Output directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory)
  + [Output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file)
  + [Log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file)
  + [Report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file)
  + [XUnit compatible result file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#xunit-compatible-result-file)
  + [Debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file)
  + [Timestamping output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamping-output-files)
  + [Setting titles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-titles)
  + [Setting background colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-background-colors)
* [3.6.2   Log levels](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels)
  + [Available log levels](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#available-log-levels)
  + [Setting log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-log-level)
  + [Visible log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#visible-log-level)
* [3.6.3   Splitting logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#splitting-logs)
* [3.6.4   Configuring statistics](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-statistics)
  + [Configuring displayed suite statistics](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-displayed-suite-statistics)
  + [Including and excluding tag statistics](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#including-and-excluding-tag-statistics)
  + [Generating combined tag statistics](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generating-combined-tag-statistics)
  + [Creating links from tag names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-links-from-tag-names)
  + [Adding documentation to tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#adding-documentation-to-tags)
* [3.6.5   Removing and flattening keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords)
  + [Removing keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords)
  + [Flattening keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#flattening-keywords)
* [3.6.6   Automatically expanding keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatically-expanding-keywords)
* [3.6.7   Setting start and end time of execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-start-and-end-time-of-execution)
* [3.6.8   Limiting error message length in reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-error-message-length-in-reports)
* [3.6.9   Programmatic modification of results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-results)
* [3.6.10   System log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log)

[**3.6.1   Different output files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-471)

This section explains what different output files can be created and how to configure where they are created. Output files are configured using command line options, which get the path to the output file in question as an argument. A special value NONE (case-insensitive) can be used to disable creating a certain output file.

[**Output directory**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-472)

All output files can be set using an absolute path, in which case they are created to the specified place, but in other cases, the path is considered relative to the output directory. The default output directory is the directory where the execution is started from, but it can be altered with the --outputdir (-d) option. The path set with this option is, again, relative to the execution directory, but can naturally be given also as an absolute path. Regardless of how a path to an individual output file is obtained, its parent directory is created automatically, if it does not exist already.

[**Output file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-473)

Output files contain all the test execution results in machine readable XML format. [Log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file), [report](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) and [xUnit](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit) files are typically generated based on them, and they can also be combined and otherwise post-processed with [Rebot](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "rebot).

**Tip**

Generating [report](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) and [xUnit](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit) files as part of test execution does not require processing output files. Disabling [log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) generation when running tests can thus save memory.

The command line option --output (-o) determines the path where the output file is created relative to the [output directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory). The default name for the output file, when tests are run, is *output.xml*.

When [post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs) with Rebot, new output files are not created unless the --output option is explicitly used.

It is possible to disable creation of the output file when running tests by giving a special value NONE to the --output option. If no outputs are needed, they should all be explicitly disabled using --output NONE --report NONE --log NONE.

[**Log file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-474)

Log files contain details about the executed test cases in HTML format. They have a hierarchical structure showing test suite, test case and keyword details. Log files are needed nearly every time when test results are to be investigated in detail. Even though log files also have statistics, reports are better for getting an higher-level overview.

The command line option --log (-l) determines where log files are created. Unless the special value NONE is used, log files are always created and their default name is *log.html*.

[테이블이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/log_passed.html)

*An example of beginning of a log file*

[테이블이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/log_failed.html)

*An example of a log file with keyword details visible*

[테이블이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/log_skipped.html)

*An example of a log file with skipped and passed tests*

[**Report file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-475)

Report files contain an overview of the test execution results in HTML format. They have statistics based on tags and executed test suites, as well as a list of all executed test cases. When both reports and logs are generated, the report has links to the log file for easy navigation to more detailed information. It is easy to see the overall test execution status from report, because its background color is green, if all tests pass and bright red if any test fails. Background can also be yellow, which means that all tests were [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped).

The command line option --report (-r) determines where report files are created. Similarly as log files, reports are always created unless NONE is used as a value, and their default name is *report.html*.

[테이블이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/report_passed.html)

*An example report file of successful test execution*

[테이블이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/report_failed.html)

*An example report file of failed test execution*

[**XUnit compatible result file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-476)

XUnit result files contain the test execution summary in [xUnit](http://en.wikipedia.org/wiki/XUnit) compatible XML format. These files can thus be used as an input for external tools that understand xUnit reports. For example, [Jenkins](http://jenkins-ci.org/) continuous integration server supports generating statistics based on xUnit compatible results.

**Tip**

Jenkins also has a separate [Robot Framework plugin](https://wiki.jenkins-ci.org/display/JENKINS/Robot+Framework+Plugin).

XUnit output files are not created unless the command line option --xunit (-x) is used explicitly. This option requires a path to the generated xUnit file, relatively to the [output directory](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory), as a value.

XUnit output files were changed pretty heavily in Robot Framework 5.0. They nowadays contain separate <testsuite> elements for each suite, <testsuite> elements have timestamp attribute, and [test suite documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation) and [metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) is stored as <property> elements.

[**Debug file**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-477)

Debug files are plain text files that are written during the test execution. All messages got from test libraries are written to them, as well as information about started and ended test suites, test cases and keywords. Debug files can be used for monitoring the test execution. This can be done using, for example, a separate [fileviewer.py](https://bitbucket.org/robotframework/robottools/src/master/fileviewer/) tool, or in UNIX-like systems, simply with the tail -f command.

Debug files are not created unless the command line option --debugfile (-b) is used explicitly.

[**Timestamping output files**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-478)

All output files listed in this section can be automatically timestamped with the option --timestampoutputs (-T). When this option is used, a timestamp in the format YYYYMMDD-hhmmss is placed between the extension and the base name of each file. The example below would, for example, create such output files as *output-20080604-163225.xml* and *mylog-20080604-163225.html*:

robot --timestampoutputs --log mylog.html --report NONE tests.robot

[**Setting titles**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-479)

The default titles for [logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) and [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) are generated by prefixing the name of the top-level test suite with *Test Log* or *Test Report*. Custom titles can be given from the command line using the options --logtitle and --reporttitle, respectively.

Example:

robot --logtitle "Smoke Test Log" --reporttitle "Smoke Test Report" --include smoke my\_tests/

**Note**

Prior to Robot Framework 3.1, underscores in the given titles were converted to spaces. Nowadays spaces need to be escaped or quoted like in the example above.

[**Setting background colors**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-480)

By default the [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) has a green background when all the tests pass, yellow background when all the test have been skipped and a red background if there are any test failrues. These colors can be customized by using the --reportbackground command line option, which takes two or three colors separated with a colon as an argument:

--reportbackground blue:red

--reportbackground green:yellow:red

--reportbackground #00E:#E00

If you specify two colors, the first one will be used instead of the default green color and the second instead of the default red. This allows, for example, using blue instead of green to make backgrounds easier to separate for color blind people.

If you specify three colors, the first one will be used when all the tests pass, the second when all tests have been skipped, and the last when there are any failures.

The specified colors are used as a value for the body element's background CSS property. The value is used as-is and can be a HTML color name (e.g. red), a hexadecimal value (e.g. #f00 or #ff0000), or an RGB value (e.g. rgb(255,0,0)). The default green and red colors are specified using hexadecimal values #9e9 and #f66, respectively.

[**3.6.2   Log levels**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-481)

[**Available log levels**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-482)

Messages in [log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) can have different log levels. Some of the messages are written by Robot Framework itself, but also executed keywords can [log information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-information) using different levels. The available log levels are:

FAIL

Used when a keyword fails. Can be used only by Robot Framework itself.

WARN

Used to display warnings. They shown also in [the console and in the Test Execution Errors section in log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution), but they do not affect the test case status.

INFO

The default level for normal messages. By default, messages below this level are not shown in the log file.

DEBUG

Used for debugging purposes. Useful, for example, for logging what libraries are doing internally. When a keyword fails, a traceback showing where in the code the failure occurred is logged using this level automatically.

TRACE

More detailed debugging level. The keyword arguments and return values are automatically logged using this level.

[**Setting log level**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-483)

By default, log messages below the INFO level are not logged, but this threshold can be changed from the command line using the --loglevel (-L) option. This option takes any of the available log levels as an argument, and that level becomes the new threshold level. A special value NONE can also be used to disable logging altogether.

It is possible to use the --loglevel option also when [post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs) with Rebot. This allows, for example, running tests initially with the TRACE level, and generating smaller log files for normal viewing later with the INFO level. By default all the messages included during execution will be included also with Rebot. Messages ignored during the execution cannot be recovered.

Another possibility to change the log level is using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Set Log Level* in the test data. It takes the same arguments as the --loglevel option, and it also returns the old level so that it can be restored later, for example, in a [test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown).

[**Visible log level**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-484)

If the log file contains messages at DEBUG or TRACE levels, a visible log level drop down is shown in the upper right corner. This allows users to remove messages below chosen level from the view. This can be useful especially when running test at TRACE level.

[텍스트이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/visible_log_level.html)

*An example log showing the visible log level drop down*

By default the drop down will be set at the lowest level in the log file, so that all messages are shown. The default visible log level can be changed using --loglevel option by giving the default after the normal log level separated by a colon:

--loglevel DEBUG:INFO

In the above example, tests are run using level DEBUG, but the default visible level in the log file is INFO.

[**3.6.3   Splitting logs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-485)

Normally the log file is just a single HTML file. When the amount of the test cases increases, the size of the file can grow so large that opening it into a browser is inconvenient or even impossible. Hence, it is possible to use the --splitlog option to split parts of the log into external files that are loaded transparently into the browser when needed.

The main benefit of splitting logs is that individual log parts are so small that opening and browsing the log file is possible even if the amount of the test data is very large. A small drawback is that the overall size taken by the log file increases.

Technically the test data related to each test case is saved into a JavaScript file in the same folder as the main log file. These files have names such as *log-42.js* where *log* is the base name of the main log file and *42* is an incremented index.

**Note**

When copying the log files, you need to copy also all the *log-\*.js* files or some information will be missing.

[**3.6.4   Configuring statistics**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-486)

There are several command line options that can be used to configure and adjust the contents of the *Statistics by Tag*, *Statistics by Suite* and *Test Details by Tag* tables in different output files. All these options work both when executing test cases and when post-processing outputs.

[**Configuring displayed suite statistics**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-487)

When a deeper suite structure is executed, showing all the test suite levels in the *Statistics by Suite* table may make the table somewhat difficult to read. By default all suites are shown, but you can control this with the command line option --suitestatlevel which takes the level of suites to show as an argument:

--suitestatlevel 3

[**Including and excluding tag statistics**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-488)

When many tags are used, the *Statistics by Tag* table can become quite congested. If this happens, the command line options --tagstatinclude and --tagstatexclude can be used to select which tags to display, similarly as --include and --exclude are used to [select test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names):

--tagstatinclude some-tag --tagstatinclude another-tag

--tagstatexclude owner-\*

--tagstatinclude prefix-\* --tagstatexclude prefix-13

[**Generating combined tag statistics**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-489)

The command line option --tagstatcombine can be used to generate aggregate tags that combine statistics from multiple tags. The combined tags are specified using [tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns) where \* and ? are supported as wildcards and AND, OR and NOT operators can be used for combining individual tags or patterns together.

The following examples illustrate creating combined tag statistics using different patterns, and the figure below shows a snippet of the resulting *Statistics by Tag* table:

--tagstatcombine owner-\*

--tagstatcombine smokeANDmytag

--tagstatcombine smokeNOTowner-janne\*



*Examples of combined tag statistics*

As the above example illustrates, the name of the added combined statistic is, by default, just the given pattern. If this is not good enough, it is possible to give a custom name after the pattern by separating them with a colon (:):

--tagstatcombine "prio1ORprio2:High priority tests"

**Note**

Prior to Robot Framework 3.1, underscores in the custom name were converted to spaces. Nowadays spaces need to be escaped or quoted like in the example above.

[**Creating links from tag names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-490)

You can add external links to the *Statistics by Tag* table by using the command line option --tagstatlink. Arguments to this option are given in the format tag:link:name, where tag specifies the tags to assign the link to, link is the link to be created, and name is the name to give to the link.

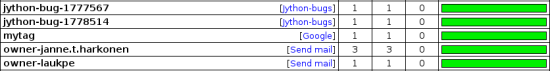
tag may be a single tag, but more commonly a [simple pattern](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns) where \* matches anything and ? matches any single character. When tag is a pattern, the matches to wildcards may be used in link and title with the syntax %N, where "N" is the index of the match starting from 1.

The following examples illustrate the usage of this option, and the figure below shows a snippet of the resulting *Statistics by Tag* table when example test data is executed with these options:

--tagstatlink mytag:http://www.google.com:Google

--tagstatlink example-bug-\*:http://example.com

--tagstatlink owner-\*:mailto:%1@domain.com?subject=Acceptance\_Tests:Send\_Mail



*Examples of links from tag names*

[**Adding documentation to tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-491)

Tags can be given a documentation with the command line option --tagdoc, which takes an argument in the format tag:doc. tag is the name of the tag to assign the documentation to, and it can also be a [simple pattern](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns) matching multiple tags. doc is the assigned documentation. It can contain simple [HTML formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting).

The given documentation is shown with matching tags in the *Test Details by Tag* table, and as a tool tip for these tags in the *Statistics by Tag* table. If one tag gets multiple documentations, they are combined together and separated with an ampersand.

Examples:

--tagdoc mytag:Example

--tagdoc "regression:\*See\* http://info.html"

--tagdoc "owner-\*:Original author"

**Note**

Prior to Robot Framework 3.1, underscores in the documentation were converted to spaces. Nowadays spaces need to be escaped or quoted like in the examples above.

[**3.6.5   Removing and flattening keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-492)

Most of the content of [output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) comes from keywords and their log messages. When creating higher level reports, log files are not necessarily needed at all, and in that case keywords and their messages just take space unnecessarily. Log files themselves can also grow overly large, especially if they contain [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) or other constructs that repeat certain keywords multiple times.

In these situations, command line options --removekeywords and --flattenkeywords can be used to dispose or flatten unnecessary keywords. They can be used both when [executing test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-test-cases-1) and when [post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs). When used during execution, they only affect the log file, not the XML output file. With rebot they affect both logs and possibly generated new output XML files.

[**Removing keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-493)

The --removekeywords option removes keywords and their messages altogether. It has the following modes of operation, and it can be used multiple times to enable multiple modes. Keywords that contain [errors or warnings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings) are not removed except when using the ALL mode.

ALL

Remove data from all keywords unconditionally.

PASSED

Remove keyword data from passed test cases. In most cases, log files created using this option contain enough information to investigate possible failures.

FOR

Remove all passed iterations from [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) except the last one.

WHILE

Remove all passed iterations from [WHILE loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#while-loops) except the last one.

WUKS

Remove all failing keywords inside [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Wait Until Keyword Succeeds* except the last one.

NAME:<pattern>

Remove data from all keywords matching the given pattern regardless the keyword status. The pattern is matched against the full name of the keyword, prefixed with the possible library or resource file name like MyLibrary.Keyword Name. The pattern is case, space, and underscore insensitive, and it supports [simple patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#simple-patterns) with \*, ? and [] as wildcards.

TAG:<pattern>

Remove data from keywords with tags that match the given pattern. Tags are case and space insensitive and they can be specified using [tag patterns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tag-patterns) where \*, ? and [] are supported as wildcards and AND, OR and NOT operators can be used for combining individual tags or patterns together. Can be used both with [library keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags) and [user keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags).

Examples:

rebot --removekeywords all --output removed.xml output.xml

robot --removekeywords passed --removekeywords for tests.robot

robot --removekeywords name:HugeKeyword --removekeywords name:resource.\* tests.robot

robot --removekeywords tag:huge tests.robot

Removing keywords is done after parsing the [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) and generating an internal model based on it. Thus it does not reduce memory usage as much as [flattening keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#flattening-keywords).

[**Flattening keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-494)

The --flattenkeywords option flattens matching keywords. In practice this means that matching keywords get all log messages from their child keywords, recursively, and child keywords are discarded otherwise. Flattening supports the following modes:

FOR

Flatten [FOR loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#for-loops) fully.

WHILE

Flatten [WHILE loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#while-loops) fully.

ITERATION

Flatten individual FOR and WHILE loop iterations.

FORITEM

Deprecated alias for ITERATION.

NAME:<pattern>

Flatten keywords matching the given pattern. Pattern matching rules are same as when [removing keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords) using the NAME:<pattern> mode.

TAG:<pattern>

Flatten keywords with tags matching the given pattern. Pattern matching rules are same as when [removing keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords) using the TAG:<pattern> mode.

Examples:

robot --flattenkeywords name:HugeKeyword --flattenkeywords name:resource.\* tests.robot

rebot --flattenkeywords foritem --output flattened.xml original.xml

Flattening keywords is done already when the [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) is parsed initially. This can save a significant amount of memory especially with deeply nested keyword structures.

[**3.6.6   Automatically expanding keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-495)

Keywords that have passed are closed in the log file by default. Thus information they contain is not visible unless you expand them. If certain keywords have important information that should be visible when the log file is opened, you can use the --expandkeywords option to set keywords automatically expanded in log file similar to failed keywords. Expanding supports the following modes:

NAME:<pattern>

Expand keywords matching the given pattern. Pattern matching rules are same as when [removing keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords) using the NAME:<pattern> mode.

TAG:<pattern>

Expand keywords with tags matching the given pattern. Pattern matching rules are same as when [removing keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-keywords) using the TAG:<pattern> mode.

If you need to expand keywords matching different names or patterns, you can use the --expandkeywords multiple times.

Examples:

robot --expandkeywords name:SeleniumLibrary.CapturePageScreenshot tests.robot

rebot --expandkeywords tag:example --expandkeywords tag:another output.xml

**Note**

The --expandkeywords option is new in Robot Framework 3.2.

[**3.6.7   Setting start and end time of execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-496)

When [combining outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#combining-outputs) using Rebot, it is possible to set the start and end time of the combined test suite using the options --starttime and --endtime, respectively. This is convenient, because by default, combined suites do not have these values. When both the start and end time are given, the elapsed time is also calculated based on them. Otherwise the elapsed time is got by adding the elapsed times of the child test suites together.

It is also possible to use the above mentioned options to set start and end times for a single suite when using Rebot. Using these options with a single output always affects the elapsed time of the suite.

Times must be given as timestamps in the format YYYY-MM-DD hh:mm:ss.mil, where all separators are optional and the parts from milliseconds to hours can be omitted. For example, 2008-06-11 17:59:20.495 is equivalent both to 20080611-175920.495 and 20080611175920495, and also mere 20080611 would work.

Examples:

rebot --starttime 20080611-17:59:20.495 output1.xml output2.xml

rebot --starttime 20080611-175920 --endtime 20080611-180242 \*.xml

rebot --starttime 20110302-1317 --endtime 20110302-11418 myoutput.xml

[**3.6.8   Limiting error message length in reports**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-497)

If a test case fails and has a long error message, the message shown in [reports](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) is automatically cut from the middle to keep reports easier to read. By default messages longer than 40 lines are cut, but that can be configured by using the --maxerrorlines command line option. The minimum value for this option is 10, and it is also possible to use a special value NONE to show the full message.

Full error messages are always visible in [log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) as messages of the failed keywords.

**Note**

The --maxerrorlines option is new in Robot Framework 3.1.

[**3.6.9   Programmatic modification of results**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-498)

If the provided built-in features to modify results are not enough, Robot Framework makes it possible to do custom modifications programmatically. This is accomplished by creating a model modifier and activating it using the --prerebotmodifier option.

This functionality works nearly exactly like [programmatic modification of test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-test-data) that can be enabled with the --prerunmodifier option. The obvious difference is that this time modifiers operate with the [result model](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#module-robot.result.model), not the [running model](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#module-robot.running.model). For example, the following modifier marks all passed tests that have taken more time than allowed as failed:

**from** **robot.api** **import** SuiteVisitor

**class** **ExecutionTimeChecker**(SuiteVisitor):

**def** \_\_init\_\_(self, max\_seconds: float):

self.max\_milliseconds = max\_seconds \* 1000

**def** visit\_test(self, test):

**if** test.status == 'PASS' **and** test.elapsedtime > self.max\_milliseconds:

test.status = 'FAIL'

test.message = 'Test execution took too long.'

If the above modifier would be in file *ExecutionTimeChecker.py*, it could be used, for example, like this:

# Specify modifier as a path when running tests. Maximum time is 42 seconds.

robot --prerebotmodifier path/to/ExecutionTimeChecker.py:42 tests.robot

# Specify modifier as a name when using Rebot. Maximum time is 3.14 seconds.

# ExecutionTimeChecker.py must be in the module search path.

rebot --prerebotmodifier ExecutionTimeChecker:3.14 output.xml

If more than one model modifier is needed, they can be specified by using the --prerebotmodifier option multiple times. When executing tests, it is possible to use --prerunmodifier and --prerebotmodifier options together.

**Note**

Argument conversion based on type hints like max\_seconds: float in the above example is new in Robot Framework 4.0 and requires Python 3.

[**3.6.10   System log**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-499)

Robot Framework has its own plain-text system log where it writes information about

* Processed and skipped test data files
* Imported test libraries, resource files and variable files
* Executed test suites and test cases
* Created outputs

Normally users never need this information, but it can be useful when investigating problems with test libraries or Robot Framework itself. A system log is not created by default, but it can be enabled by setting the environment variable ROBOT\_SYSLOG\_FILE so that it contains a path to the selected file.

A system log has the same [log levels](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels) as a normal log file, with the exception that instead of FAIL it has the ERROR level. The threshold level to use can be altered using the ROBOT\_SYSLOG\_LEVEL environment variable like shown in the example below. Possible [unexpected errors and warnings](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) are written into the system log in addition to the console and the normal log file.

*#!/bin/bash*

export ROBOT\_SYSLOG\_FILE=/tmp/syslog.txt

export ROBOT\_SYSLOG\_LEVEL=DEBUG

robot --name Syslog\_example path/to/tests

[**4   Extending Robot Framework**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-118)

* [4.1   Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries)
* [4.2   Remote library interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library-interface)
* [4.3   Listener interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface)

[**4.1   Creating test libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-500)

Robot Framework's actual testing capabilities are provided by test libraries. There are many existing libraries, some of which are even bundled with the core framework, but there is still often a need to create new ones. This task is not too complicated because, as this chapter illustrates, Robot Framework's library API is simple and straightforward.

* [4.1.1   Introduction](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#introduction-2)
  + [Supported programming languages](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-programming-languages)
  + [Different test library APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-test-library-apis)
* [4.1.2   Creating test library class or module](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-library-class-or-module)
  + [Library name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-name)
  + [Providing arguments to libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#providing-arguments-to-libraries)
  + [Library scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope)
  + [Library version](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-version)
  + [Documentation format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-format)
  + [Library acting as listener](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-acting-as-listener)
  + [@library decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-decorator)
* [4.1.3   Creating keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-keywords)
  + [What methods are considered keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#what-methods-are-considered-keywords)
  + [Limiting public methods becoming keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-public-methods-becoming-keywords)
  + [Keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-names)
  + [Keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags)
  + [Keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments)
  + [Default values to keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values-to-keywords)
  + [Variable number of arguments (\*varargs)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments-varargs)
  + [Free keyword arguments (\*\*kwargs)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-keyword-arguments-kwargs)
  + [Keyword-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-only-arguments)
  + [Positional-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-only-arguments)
  + [Argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-conversion)
  + [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator)
  + [@not\_keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#not-keyword-decorator)
  + [Using custom decorators](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-custom-decorators)
  + [Embedding arguments into keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-names)
* [4.1.4   Communicating with Robot Framework](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#communicating-with-robot-framework)
  + [Reporting keyword status](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reporting-keyword-status)
  + [Exceptions provided by Robot Framework](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#exceptions-provided-by-robot-framework)
  + [Continuable failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continuable-failures)
  + [Skipping tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipping-tests)
  + [Stopping test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution)
  + [Logging information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-information)
  + [Programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis)
  + [Logging during library initialization](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-during-library-initialization)
  + [Returning values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#returning-values)
  + [Communication when using threads](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#communication-when-using-threads)
* [4.1.5   Distributing test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#distributing-test-libraries)
  + [Documenting libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-libraries)
  + [Testing libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#testing-libraries)
  + [Packaging libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#packaging-libraries)
  + [Deprecating keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#deprecating-keywords)
* [4.1.6   Dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api)
  + [Getting keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-names)
  + [Running keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-keywords)
  + [Getting keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments)
  + [Getting keyword argument types](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-argument-types)
  + [Getting keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-tags)
  + [Getting keyword documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-documentation)
  + [Getting general library documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-general-library-documentation)
  + [Getting keyword source information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-source-information)
  + [Named argument syntax with dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax-with-dynamic-libraries)
  + [Free named arguments with dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-dynamic-libraries)
  + [Named-only arguments with dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-dynamic-libraries)
  + [Summary](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#summary)
* [4.1.7   Hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api)
  + [Getting keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-names-1)
  + [Running keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-keywords-1)
  + [Getting keyword arguments and documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments-and-documentation)
  + [Summary](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#summary-1)
* [4.1.8   Using Robot Framework's internal modules](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-robot-framework-s-internal-modules)
  + [Available APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#available-apis)
  + [Using BuiltIn library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-builtin-library)
* [4.1.9   Extending existing test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extending-existing-test-libraries)
  + [Modifying original source code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#modifying-original-source-code)
  + [Using inheritance](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-inheritance)
  + [Using other libraries directly](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-other-libraries-directly)
  + [Getting active library instance from Robot Framework](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-active-library-instance-from-robot-framework)
  + [Libraries using dynamic or hybrid API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libraries-using-dynamic-or-hybrid-api)

[**4.1.1   Introduction**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-503)

[**Supported programming languages**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-504)

Robot Framework itself is written with [Python](http://python.org/) and naturally test libraries extending it can be implemented using the same language. It is also possible to implement libraries with C using [Python C API](http://docs.python.org/c-api/index.html), although it is often easier to interact with C code from Python libraries using [ctypes](http://docs.python.org/library/ctypes.html) module.

Libraries implemented using Python can also act as wrappers to functionality implemented using other programming languages. A good example of this approach is the [Remote library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library), and another widely used approaches is running external scripts or tools as separate processes.

[**Different test library APIs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-505)

Robot Framework has three different test library APIs.

Static API

The simplest approach is having a module or a class with functions/methods which map directly to [keyword names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-names). Keywords also take the same [arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments) as the methods implementing them. Keywords [report failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reporting-keyword-status) with exceptions, [log](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-information) by writing to standard output and can [return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#returning-values) using the return statement.

Dynamic API

Dynamic libraries are classes that implement a method to get the names of the keywords they implement, and another method to execute a named keyword with given arguments. The names of the keywords to implement, as well as how they are executed, can be determined dynamically at runtime, but reporting the status, logging and returning values is done similarly as in the static API.

Hybrid API

This is a hybrid between the static and the dynamic API. Libraries are classes with a method telling what keywords they implement, but those keywords must be available directly. Everything else except discovering what keywords are implemented is similar as in the static API.

All these APIs are described in this chapter. Everything is based on how the static API works, so its functions are discussed first. How the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) and the [hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api) differ from it is then discussed in sections of their own.

[**4.1.2   Creating test library class or module**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-506)

Test libraries can be implemented as Python modules or classes.

[**Library name**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-507)

The name of a test library that is used when a library is imported is the same as the name of the module or class implementing it. For example, if you have a Python module MyLibrary (that is, file *MyLibrary.py*), it will create a library with name *MyLibrary*.

Python classes are always inside a module. If the name of a class implementing a library is the same as the name of the module, Robot Framework allows dropping the class name when importing the library. For example, class MyLib in *MyLib.py* file can be used as a library with just name *MyLib*. This also works with submodules so that if, for example, parent.MyLib module has class MyLib, importing it using just *parent.MyLib* works. If the module name and class name are different, libraries must be taken into use using both module and class names, such as *mymodule.MyLibrary* or *parent.submodule.MyLib*.

**Tip**

If the library name is really long, it is recommended to give the library a simpler alias by using the [WITH NAME syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name-to-test-library).

[**Providing arguments to libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-508)

All test libraries implemented as classes can take arguments. These arguments are specified in the Setting section after the library name, and when Robot Framework creates an instance of the imported library, it passes them to its constructor. Libraries implemented as a module cannot take any arguments, so trying to use those results in an error.

The number of arguments needed by the library is the same as the number of arguments accepted by the library's constructor. The default values and variable number of arguments work similarly as with [keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments). Arguments passed to the library, as well as the library name itself, can be specified using variables, so it is possible to alter them, for example, from the command line.

**\*\*\* Settings \*\*\***

**Library** **MyLibrary** 10.0.0.1 8080

**Library** **AnotherLib** ${VAR}

Example implementations for the libraries used in the above example:

**from** **example** **import** Connection

**class** **MyLibrary**:

**def** \_\_init\_\_(self, host, port=80):

self.\_conn = Connection(host, int(port))

**def** send\_message(self, message):

self.\_conn.send(message)

**class** **AnotherLib**:

**def** \_\_init\_\_(self, environment):

self.environment = environment

**def** do\_something(self):

**if** self.environment == 'test':

*# do something in test environment*

**else**:

*# do something in other environments*

[**Library scope**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-509)

Libraries implemented as classes can have an internal state, which can be altered by keywords and with arguments to the constructor of the library. Because the state can affect how keywords actually behave, it is important to make sure that changes in one test case do not accidentally affect other test cases. These kind of dependencies may create hard-to-debug problems, for example, when new test cases are added and they use the library inconsistently.

Robot Framework attempts to keep test cases independent from each other: by default, it creates new instances of test libraries for every test case. However, this behavior is not always desirable, because sometimes test cases should be able to share a common state. Additionally, all libraries do not have a state and creating new instances of them is simply not needed.

Test libraries can control when new libraries are created with a class attribute ROBOT\_LIBRARY\_SCOPE . This attribute must be a string and it can have the following three values:

TEST

A new instance is created for every test case. A possible suite setup and suite teardown share yet another instance.

Prior to Robot Framework 3.2 this value was TEST CASE, but nowadays TEST is recommended. Because all unrecognized values are considered same as TEST, both values work with all versions. For the same reason it is possible to also use value TASK if the library is targeted for [RPA](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#rpa) usage more than testing. TEST is also the default value if the ROBOT\_LIBRARY\_SCOPE attribute is not set.

SUITE

A new instance is created for every test suite. The lowest-level test suites, created from test case files and containing test cases, have instances of their own, and higher-level suites all get their own instances for their possible setups and teardowns.

Prior to Robot Framework 3.2 this value was TEST SUITE. That value still works, but SUITE is recommended with libraries targeting Robot Framework 3.2 and newer.

GLOBAL

Only one instance is created during the whole test execution and it is shared by all test cases and test suites. Libraries created from modules are always global.

**Note**

If a library is imported multiple times with different [arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#providing-arguments-to-libraries), a new instance is created every time regardless the scope.

When the SUITE or GLOBAL scopes are used with libraries that have a state, it is recommended that libraries have some special keyword for cleaning up the state. This keyword can then be used, for example, in a suite setup or teardown to ensure that test cases in the next test suites can start from a known state. For example, *SeleniumLibrary* uses the GLOBAL scope to enable using the same browser in different test cases without having to reopen it, and it also has the *Close All Browsers* keyword for easily closing all opened browsers.

Example library using the SUITE scope:

**class** **ExampleLibrary**:

ROBOT\_LIBRARY\_SCOPE = 'SUITE'

**def** \_\_init\_\_(self):

self.\_counter = 0

**def** count(self):

self.\_counter += 1

print(self.\_counter)

**def** clear\_counter(self):

self.\_counter = 0

[**Library version**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-510)

When a test library is taken into use, Robot Framework tries to determine its version. This information is then written into the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) to provide debugging information. Library documentation tool [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) also writes this information into the keyword documentations it generates.

Version information is read from attribute ROBOT\_LIBRARY\_VERSION, similarly as [library scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope) is read from ROBOT\_LIBRARY\_SCOPE. If ROBOT\_LIBRARY\_VERSION does not exist, information is tried to be read from \_\_version\_\_ attribute. These attributes must be class or module attributes, depending whether the library is implemented as a class or a module.

An example module using \_\_version\_\_:

\_\_version\_\_ = '0.1'

**def** keyword():

**pass**

[**Documentation format**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-511)

Library documentation tool [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) supports documentation in multiple formats. If you want to use something else than Robot Framework's own [documentation formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting), you can specify the format in the source code using ROBOT\_LIBRARY\_DOC\_FORMAT attribute similarly as [scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope) and [version](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-version) are set with their own ROBOT\_LIBRARY\_\* attributes.

The possible case-insensitive values for documentation format are ROBOT (default), HTML, TEXT (plain text), and reST ([reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText)). Using the reST format requires the [docutils](https://pypi.python.org/pypi/docutils) module to be installed when documentation is generated.

Setting the documentation format is illustrated by the following example that uses reStructuredText format. See [Documenting libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-libraries) section and [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) chapter for more information about documenting test libraries in general.

*"""A library for \*documentation format\* demonstration purposes.*

*This documentation is created using reStructuredText\_\_. Here is a link*

*to the only \`Keyword\`.*

*\_\_ http://docutils.sourceforge.net*

*"""*

ROBOT\_LIBRARY\_DOC\_FORMAT = 'reST'

**def** keyword():

*"""\*\*Nothing\*\* to see here. Not even in the table below.*

*======= ===== =====*

*Table here has*

*nothing to see.*

*======= ===== =====*

*"""*

**pass**

[**Library acting as listener**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-512)

[Listener interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface) allows external listeners to get notifications about test execution. They are called, for example, when suites, tests, and keywords start and end. Sometimes getting such notifications is also useful for test libraries, and they can register a custom listener by using ROBOT\_LIBRARY\_LISTENER attribute. The value of this attribute should be an instance of the listener to use, possibly the library itself.

For more information and examples see [Libraries as listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libraries-as-listeners) section.

[**@library decorator**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-513)

An easy way to configure libraries implemented as classes is using the robot.api.deco.library class decorator. It allows configuring library's [scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope), [version](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-version), [custom argument converters](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#custom-argument-converters), [documentation format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-format) and [listener](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-acting-as-listener) with optional arguments scope, version, converter, doc\_format and listener, respectively. When these arguments are used, they set the matching ROBOT\_LIBRARY\_SCOPE, ROBOT\_LIBRARY\_VERSION, ROBOT\_LIBRARY\_CONVERTERS, ROBOT\_LIBRARY\_DOC\_FORMAT and ROBOT\_LIBRARY\_LISTENER attributes automatically:

**from** **robot.api.deco** **import** library

**from** **example** **import** Listener

@library(scope='GLOBAL', version='3.2b1', doc\_format='reST', listener=Listener())

**class** **Example**:

*# ...*

The @library decorator also disables the [automatic keyword discovery](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#what-methods-are-considered-keywords) by setting the ROBOT\_AUTO\_KEYWORDS argument to False by default. This means that it is mandatory to decorate methods with the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator) to expose them as keywords. If only that behavior is desired and no further configuration is needed, the decorator can also be used without parenthesis like:

**from** **robot.api.deco** **import** library

@library

**class** **Example**:

*# ...*

If needed, the automatic keyword discovery can be enabled by using the auto\_keywords argument:

**from** **robot.api.deco** **import** library

@library(scope='GLOBAL', auto\_keywords=**True**)

**class** **Example**:

*# ...*

The @library decorator only sets class attributes ROBOT\_LIBRARY\_SCOPE, ROBOT\_LIBRARY\_VERSION, ROBOT\_LIBRARY\_CONVERTERS, ROBOT\_LIBRARY\_DOC\_FORMAT and ROBOT\_LIBRARY\_LISTENER if the respective arguments scope, version, converters, doc\_format and listener are used. The ROBOT\_AUTO\_KEYWORDS attribute is set always. When attributes are set, they override possible existing class attributes.

**Note**

The @library decorator is new in Robot Framework 3.2 and converters argument is new in Robot Framework 5.0.

[**4.1.3   Creating keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-514)

[**What methods are considered keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-515)

When the static library API is used, Robot Framework uses introspection to find out what keywords the library class or module implements. By default it excludes methods and functions starting with an underscore. All the methods and functions that are not ignored are considered keywords. For example, the library below implements a single keyword *My Keyword*.

**class** **MyLibrary**:

**def** my\_keyword(self, arg):

**return** self.\_helper\_method(arg)

**def** \_helper\_method(self, arg):

**return** arg.upper()

[**Limiting public methods becoming keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-516)

Automatically considering all public methods and functions keywords typically works well, but there are cases where it is not desired. There are also situations where keywords are created when not expected. For example, when implementing a library as class, it can be a surprise that also methods in possible base classes are considered keywords. When implementing a library as a module, functions imported into the module namespace becoming keywords is probably even a bigger surprise.

This section explains how to prevent methods and functions becoming keywords.

**Class based libraries**

When a library is implemented as a class, it is possible to tell Robot Framework not to automatically expose methods as keywords by setting the ROBOT\_AUTO\_KEYWORDS attribute to the class with a false value:

**class** **Example**:

ROBOT\_AUTO\_KEYWORDS = **False**

When the ROBOT\_AUTO\_KEYWORDS attribute is set like this, only methods that have explicitly been decorated with the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator) or otherwise have the robot\_name attribute become keywords. The @keyword decorator can also be used for setting a [custom name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name), [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags) and [argument types](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator) to the keyword.

Although the ROBOT\_AUTO\_KEYWORDS attribute can be set to the class explicitly, it is more convenient to use the [@library decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-decorator) that sets it to False by default:

**from** **robot.api.deco** **import** keyword, library

@library

**class** **Example**:

@keyword

**def** this\_is\_keyword(self):

**pass**

@keyword('This is keyword with custom name')

**def** xxx(self):

**pass**

**def** this\_is\_not\_keyword(self):

**pass**

**Note**

Both limiting what methods become keywords using the ROBOT\_AUTO\_KEYWORDS attribute and the @library decorator are new in Robot Framework 3.2.

Another way to explicitly specify what keywords a library implements is using the [dynamic](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) or the [hybrid](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api) library API.

**Module based libraries**

When implementing a library as a module, all functions in the module namespace become keywords. This is true also with imported functions, and that can cause nasty surprises. For example, if the module below would be used as a library, it would contain a keyword *Example Keyword*, as expected, but also a keyword *Current Thread*.

**from** **threading** **import** current\_thread

**def** example\_keyword():

print('Running in thread "**%s**".' % current\_thread().name)

A simple way to avoid imported functions becoming keywords is to only import modules (e.g. import threading) and to use functions via the module (e.g threading.current\_thread()). Alternatively functions could be given an alias starting with an underscore at the import time (e.g. from threading import current\_thread as \_current\_thread).

A more explicit way to limit what functions become keywords is using the module level \_\_all\_\_ attribute that [Python itself uses for similar purposes](https://docs.python.org/tutorial/modules.html#importing-from-a-package). If it is used, only the listed functions can be keywords. For example, the library below implements only one keyword *Example Keyword*:

**from** **threading** **import** current\_thread

\_\_all\_\_ = ['example\_keyword']

**def** example\_keyword():

print('Running in thread "**%s**".' % current\_thread().name)

**def** this\_is\_not\_keyword():

**pass**

If the library is big, maintaining the \_\_all\_\_ attribute when keywords are added, removed or renamed can be a somewhat big task. Another way to explicitly mark what functions are keywords is using the ROBOT\_AUTO\_KEYWORDS attribute similarly as it can be used with [class based libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#class-based-libraries). When this attribute is set to a false value, only functions explicitly decorated with the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator) become keywords. For example, also this library implements only one keyword *Example Keyword*:

**from** **threading** **import** current\_thread

**from** **robot.api.deco** **import** keyword

ROBOT\_AUTO\_KEYWORDS = **False**

@keyword

**def** example\_keyword():

print('Running in thread "**%s**".' % current\_thread().name)

**def** this\_is\_not\_keyword():

**pass**

**Note**

Limiting what functions become keywords using ROBOT\_AUTO\_KEYWORDS is a new feature in Robot Framework 3.2.

**Using @not\_keyword decorator**

Functions in modules and methods in classes can be explicitly marked as "not keywords" by using the @not\_keyword decorator. When a library is implemented as a module, this decorator can also be used to avoid imported functions becoming keywords.

**from** **threading** **import** current\_thread

**from** **robot.api.deco** **import** not\_keyword

not\_keyword(current\_thread) *# Don't expose `current\_thread` as a keyword.*

**def** example\_keyword():

print('Running in thread "**%s**".' % current\_thread().name)

@not\_keyword

**def** this\_is\_not\_keyword():

**pass**

Using the @not\_keyword decorator is pretty much the opposite way to avoid functions or methods becoming keywords compared to disabling the automatic keyword discovery with the @library decorator or by setting the ROBOT\_AUTO\_KEYWORDS to a false value. Which one to use depends on the context.

**Note**

The @not\_keyword decorator is new in Robot Framework 3.2.

[**Keyword names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-517)

Keyword names used in the test data are compared with method names to find the method implementing these keywords. Name comparison is case-insensitive, and also spaces and underscores are ignored. For example, the method hello maps to the keyword name *Hello*, *hello* or even *h e l l o*. Similarly both the do\_nothing and doNothing methods can be used as the *Do Nothing* keyword in the test data.

Example library implemented as a module in the *MyLibrary.py* file:

**def** hello(name):

print("Hello, **%s**!" % name)

**def** do\_nothing():

**pass**

The example below illustrates how the example library above can be used. If you want to try this yourself, make sure that the library is in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path).

**\*\*\* Settings \*\*\***

**Library** **MyLibrary**

**\*\*\* Test Cases \*\*\***

**My Test**

Do Nothing

Hello world

**Setting custom name**

It is possible to expose a different name for a keyword instead of the default keyword name which maps to the method name. This can be accomplished by setting the robot\_name attribute on the method to the desired custom name:

**def** login(username, password):

*# ...*

login.robot\_name = 'Login via user panel'

**\*\*\* Test Cases \*\*\***

**My Test**

Login Via User Panel ${username} ${password}

Instead of explicitly setting the robot\_name attribute like in the above example, it is typically easiest to use the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator):

**from** **robot.api.deco** **import** keyword

@keyword('Login via user panel')

**def** login(username, password):

*# ...*

Using this decorator without an argument will have no effect on the exposed keyword name, but will still set the robot\_name attribute. This allows [marking methods to expose as keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#marking-methods-to-expose-as-keywords) without actually changing keyword names. Methods that have the robot\_name attribute also create keywords even if the method name itself would start with an underscore.

Setting a custom keyword name can also enable library keywords to accept arguments using the [embedded arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-names) syntax.

[**Keyword tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-518)

Library keywords and [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags) can have tags. Library keywords can define them by setting the robot\_tags attribute on the method to a list of desired tags. Similarly as when [setting custom name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name), it is easiest to set this attribute by using the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator):

**from** **robot.api.deco** **import** keyword

@keyword(tags=['tag1', 'tag2'])

**def** login(username, password):

*# ...*

@keyword('Custom name', ['tags', 'here'])

**def** another\_example():

*# ...*

Another option for setting tags is giving them on the last line of [keyword documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-libraries) with Tags: prefix and separated by a comma. For example:

**def** login(username, password):

*"""Log user in to SUT.*

*Tags: tag1, tag2*

*"""*

*# ...*

[**Keyword arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-519)

With a static and hybrid API, the information on how many arguments a keyword needs is got directly from the method that implements it. Libraries using the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) have other means for sharing this information, so this section is not relevant to them.

The most common and also the simplest situation is when a keyword needs an exact number of arguments. In this case, the method simply take exactly those arguments. For example, a method implementing a keyword with no arguments takes no arguments either, a method implementing a keyword with one argument also takes one argument, and so on.

Example keywords taking different numbers of arguments:

**def** no\_arguments():

print("Keyword got no arguments.")

**def** one\_argument(arg):

print("Keyword got one argument '**%s**'." % arg)

**def** three\_arguments(a1, a2, a3):

print("Keyword got three arguments '**%s**', '**%s**' and '**%s**'." % (a1, a2, a3))

[**Default values to keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-520)

It is often useful that some of the arguments that a keyword uses have default values.

In Python a method has always exactly one implementation and possible default values are specified in the method signature. The syntax, which is familiar to all Python programmers, is illustrated below:

**def** one\_default(arg='default'):

print("Argument has value **%s**" % arg)

**def** multiple\_defaults(arg1, arg2='default 1', arg3='default 2'):

print("Got arguments **%s**, **%s** and **%s**" % (arg1, arg2, arg3))

The first example keyword above can be used either with zero or one arguments. If no arguments are given, arg gets the value default. If there is one argument, arg gets that value, and calling the keyword with more than one argument fails. In the second example, one argument is always required, but the second and the third one have default values, so it is possible to use the keyword with one to three arguments.

**\*\*\* Test Cases \*\*\***

**Defaults**

One Default

One Default argument

Multiple Defaults required arg

Multiple Defaults required arg optional

Multiple Defaults required arg optional 1 optional 2

[**Variable number of arguments (\*varargs)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-521)

Robot Framework supports also keywords that take any number of arguments.

Python supports methods accepting any number of arguments. The same syntax works in libraries and, as the examples below show, it can also be combined with other ways of specifying arguments:

**def** any\_arguments(\*args):

print("Got arguments:")

**for** arg **in** args:

print(arg)

**def** one\_required(required, \*others):

print("Required: **%s\n**Others:" % required)

**for** arg **in** others:

print(arg)

**def** also\_defaults(req, def1="default 1", def2="default 2", \*rest):

print(req, def1, def2, rest)

**\*\*\* Test Cases \*\*\***

**Varargs**

Any Arguments

Any Arguments argument

Any Arguments arg 1 arg 2 arg 3 arg 4 arg 5

One Required required arg

One Required required arg another arg yet another

Also Defaults required

Also Defaults required these two have defaults

Also Defaults 1 2 3 4 5 6

[**Free keyword arguments (\*\*kwargs)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-522)

Robot Framework supports [Python's \*\*kwargs syntax](https://docs.python.org/tutorial/controlflow.html#keyword-arguments). How to use use keywords that accept *free keyword arguments*, also known as *free named arguments*, is [discussed under the Creating test cases section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments). In this section we take a look at how to create such keywords.

If you are already familiar how kwargs work with Python, understanding how they work with Robot Framework test libraries is rather simple. The example below shows the basic functionality:

**def** example\_keyword(\*\*stuff):

**for** name, value **in** stuff.items():

print(name, value)

**\*\*\* Test Cases \*\*\***

**Keyword Arguments**

Example Keyword hello=world *# Logs 'hello world'.*

Example Keyword foo=1 bar=42 *# Logs 'foo 1' and 'bar 42'.*

Basically, all arguments at the end of the keyword call that use the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) name=value, and that do not match any other arguments, are passed to the keyword as kwargs. To avoid using a literal value like foo=quux as a free keyword argument, it must be [escaped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) like foo\=quux.

The following example illustrates how normal arguments, varargs, and kwargs work together:

**def** various\_args(arg=**None**, \*varargs, \*\*kwargs):

**if** arg **is** **not** **None**:

print('arg:', arg)

**for** value **in** varargs:

print('vararg:', value)

**for** name, value **in** sorted(kwargs.items()):

print('kwarg:', name, value)

**\*\*\* Test Cases \*\*\***

**Positional**

Various Args hello world *# Logs 'arg: hello' and 'vararg: world'.*

**Named**

Various Args arg=value *# Logs 'arg: value'.*

**Kwargs**

Various Args a=1 b=2 c=3 *# Logs 'kwarg: a 1', 'kwarg: b 2' and 'kwarg: c 3'.*

Various Args c=3 a=1 b=2 *# Same as above. Order does not matter.*

**Positional and kwargs**

Various Args 1 2 kw=3 *# Logs 'arg: 1', 'vararg: 2' and 'kwarg: kw 3'.*

**Named and kwargs**

Various Args arg=value hello=world *# Logs 'arg: value' and 'kwarg: hello world'.*

Various Args hello=world arg=value *# Same as above. Order does not matter.*

For a real world example of using a signature exactly like in the above example, see *Run Process* and *Start Keyword* keywords in the [Process](https://robotframework.org/robotframework/latest/libraries/Process.html) library.

[**Keyword-only arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-523)

Starting from Robot Framework 3.1, it is possible to use [named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments) with different keywords. This support is provided by Python's [keyword-only arguments](https://www.python.org/dev/peps/pep-3102). Keyword-only arguments are specified after possible \*varargs or after a dedicated \* marker when \*varargs are not needed. Possible \*\*kwargs are specified after keyword-only arguments.

Example:

**def** sort\_words(\*words, case\_sensitive=**False**):

key = str.lower **if** case\_sensitive **else** **None**

**return** sorted(words, key=key)

**def** strip\_spaces(word, \*, left=**True**, right=**True**):

**if** left:

word = word.lstrip()

**if** right:

word = word.rstrip()

**return** word

**\*\*\* Test Cases \*\*\***

**Example**

Sort Words Foo bar baZ

Sort Words Foo bar baZ case\_sensitive=True

Strip Spaces ${word} left=False

[**Positional-only arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-524)

Python 3.8 introduced [positional-only arguments](https://www.python.org/dev/peps/pep-0570/) that make it possible to specify that an argument can only be given as a [positional argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-argument), not as a [named argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument) like name=value. Positional-only arguments are specified before normal arguments and a special / marker must be used after them:

**def** keyword(posonly, /, normal):

print(f"Got positional-only argument **{**posonly**}** and normal argument **{**normal**}**.")

The above keyword could be used like this:

**\*\*\* Test Cases \*\*\***

**Example**

*# Positional-only and normal argument used as positional arguments.*

Keyword foo bar

*# Normal argument can also be named.*

Keyword foo normal=bar

If a positional-only argument is used with a value that contains an equal sign like example=usage, it is not considered to mean [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) even if the part before the = would match the argument name. This rule only applies if the positional-only argument is used in its correct position without other arguments using the name argument syntax before it, though.

**\*\*\* Test Cases \*\*\***

**Example**

*# Positional-only argument gets literal value `posonly=foo` in this case.*

Keyword posonly=foo normal=bar

*# This fails.*

Keyword normal=bar posonly=foo

Positional-only arguments are fully supported starting from Robot Framework 4.0. Using them as positional arguments works also with earlier versions, but using them as named arguments causes an error on Python side.

[**Argument conversion**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-525)

Arguments defined in Robot Framework test data are, by default, passed to keywords as Unicode strings. There are, however, several ways to use non-string values as well:

* [Variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) can contain any kind of objects as values, and variables used as arguments are passed to keywords as-is.
* Keywords can themselves [convert arguments they accept](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#manual-argument-conversion) to other types.
* It is possible to specify argument types explicitly using [function annotations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations) or the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator). In these cases Robot Framework converts arguments automatically.
* Automatic conversion is also done based on [keyword default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values).
* Libraries can register [custom argument converters](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#custom-argument-converters).

Automatic argument conversion based on function annotations, types specified using the @keyword decorator, and argument default values are all new features in Robot Framework 3.1. The [Supported conversions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) section specifies which argument conversion are supported in these cases.

Prior to Robot Framework 4.0, automatic conversion was done only if the given argument was a string. Nowadays it is done regardless the argument type.

**Manual argument conversion**

If no type information is specified to Robot Framework, all arguments not passed as [variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables) are given to keywords as Unicode strings. This includes cases like this:

**\*\*\* Test Cases \*\*\***

**Example**

Example Keyword 42 False

It is always possible to convert arguments passed as strings insider keywords. In simple cases this means using int() or float() to convert arguments to numbers, but other kind of conversion is possible as well. When working with Boolean values, care must be taken because all non-empty strings, including string False, are considered true by Python. Robot Framework's own robot.utils.is\_truthy() utility handles this nicely as it considers strings like FALSE, NO and NONE (case-insensitively) to be false:

**def** example\_keyword(count, case\_insensitive=**True**):

count = int(count)

**if** is\_truthy(case\_insensitive):

*# ...*

Notice that with Robot Framework 3.1 and newer is\_truthy is not needed in the above example because argument type would be got based on the [default value](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values).

**Specifying argument types using function annotations**

Starting from Robot Framework 3.1, arguments passed to keywords are automatically converted if argument type information is available and the type is recognized. The most natural way to specify types is using Python [function annotations](https://www.python.org/dev/peps/pep-3107/). For example, the keyword in the previous example could be implemented as follows and arguments would be converted automatically:

**def** example\_keyword(count: int, case\_insensitive: bool = **True**):

**if** case\_insensitive:

*# ...*

See the [Supported conversions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) section below for a list of types that are automatically converted and what values these types accept. It is an error if an argument having one of the supported types is given a value that cannot be converted. Annotating only some of the arguments is fine.

Annotating arguments with other than the supported types is not an error, and it is also possible to use annotations for other than typing purposes. In those cases no conversion is done, but annotations are nevertheless shown in the documentation generated by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc).

**Specifying argument types using @keyword decorator**

An alternative way to specify explicit argument types is using the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator). Starting from Robot Framework 3.1, it accepts an optional types argument that can be used to specify argument types either as a dictionary mapping argument names to types or as a list mapping arguments to types based on position. These approaches are shown below implementing the same keyword as in earlier examples:

**from** **robot.api.deco** **import** keyword

@keyword(types={'count': int, 'case\_insensitive': bool})

**def** example\_keyword(count, case\_insensitive=**True**):

**if** case\_insensitive:

*# ...*

@keyword(types=[int, bool])

**def** example\_keyword(count, case\_insensitive=**True**):

**if** case\_insensitive:

*# ...*

Regardless of the approach that is used, it is not necessarily to specify types for all arguments. When specifying types as a list, it is possible to use None to mark that a certain argument does not have a type, and arguments at the end can be omitted altogether. For example, both of these keywords specify the type only for the second argument:

@keyword(types={'second': float})

**def** example1(first, second, third):

*# ...*

@keyword(types=[**None**, float])

**def** example2(first, second, third):

*# ...*

If any types are specified using the @keyword decorator, type information got from [annotations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations) is ignored with that keyword. Setting types to None like @keyword(types=None) disables type conversion altogether so that also type information got from [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values) is ignored.

**Implicit argument types based on default values**

If type information is not got explicitly using annotations or the @keyword decorator, Robot Framework 3.1 and newer tries to get it based on possible argument default value. In this example count and case\_insensitive get types int and bool, respectively:

**def** example\_keyword(count=-1, case\_insensitive=**True**):

**if** case\_insensitive:

*# ...*

When type information is got implicitly based on the default values, argument conversion itself is not as strict as when the information is got explicitly:

* Conversion may be attempted also to other "similar" types. For example, if converting to an integer fails, float conversion is attempted.
* Conversion failures are not errors, keywords get the original value in these cases instead.

If an argument has an explicit type and a default value, conversion is first attempted based on the explicit type. If that fails, then conversion is attempted based on the default value. In this special case conversion based on the default value is strict and a conversion failure causes an error.

If argument conversion based on default values is not desired, the whole argument conversion can be disabled with the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator) like @keyword(types=None).

**Note**

Prior to Robot Framework 4.0 conversion was done based on the default value only if the argument did not have an explict type.

**Supported conversions**

The table below lists the types that Robot Framework 3.1 and newer convert arguments to. These characteristics apply to all conversions:

* Type can be explicitly specified using [function annotations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations) or the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator).
* If not explicitly specified, type can be got implicitly from [argument default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values).
* Conversion is done regardless of the type of the given argument. If the argument type is incompatible with the expected type, conversion fails.
* Conversion failures cause an error if the type has been specified explicitly. If the type is got based on a default value, the given argument is used as-is.

The type to use can be specified either using concrete types (e.g. [list](https://docs.python.org/library/stdtypes.html#list)), by using Abstract Base Classes (ABC) (e.g. [Sequence](https://docs.python.org/library/collections.abc.html#collections.abc.Sequence)), or by using sub classes of these types (e.g. [MutableSequence](https://docs.python.org/library/collections.abc.html" \l "collections.abc.MutableSequence)). In all these cases the argument is converted to the concrete type.

Also types in in the [typing](https://docs.python.org/library/typing.html) module that map to the supported concrete types or ABCs (e.g. List) are supported. With generics also the subscription syntax (e.g. List[int]) works, but no validation is done for container contents.

In addition to using the actual types (e.g. int), it is possible to specify the type using type names as a string (e.g. 'int') and some types also have aliases (e.g. 'integer'). Matching types to names and aliases is case-insensitive.

The Accepts column specifies which given argument types are converted. If the given argument already has the expected type, no conversion is done. Other types cause conversion failures.

| *Supported argument conversions* | | | | | |
| --- | --- | --- | --- | --- | --- |
| **Type** | **ABC** | **Aliases** | **Accepts** | **Explanation** | **Examples** |
| [bool](https://docs.python.org/library/functions.html#bool) |  | boolean | [str](https://docs.python.org/library/functions.html#func-str), [int](https://docs.python.org/library/functions.html#int) , [float](https://docs.python.org/library/functions.html#float), [None](https://docs.python.org/library/constants.html#None) | Strings TRUE, YES, ON and 1 are converted to True, the empty string as well as FALSE, NO, OFF and 0 are converted to False, and the string NONE is converted to None. Other strings and other accepted values are passed as-is, allowing keywords to handle them specially if needed. All string comparisons are case-insensitive. | TRUE (converted to True)  off (converted to False)  example (used as-is) |
| [int](https://docs.python.org/library/functions.html#int) | [Integral](https://docs.python.org/library/numbers.html#numbers.Integral) | integer, long | [str](https://docs.python.org/library/functions.html#func-str), [float](https://docs.python.org/library/functions.html#float) | Conversion is done using the [int](https://docs.python.org/library/functions.html#int) built-in function. Floats are accepted only if they can be represented as integers exactly. For example, 1.0 is accepted and 1.1 is not. If converting a string to an integer fails and the type is got implicitly based on a default value, conversion to float is attempted as well.  Starting from RF 4.1, it is possible to use hexadecimal, octal and binary numbers by prefixing values with 0x, 0o and 0b, respectively.  Starting from RF 4.1, spaces and underscores can be used as visual separators for digit grouping purposes. | 42  -1  0xFF  0o777  0b1010  10 000 000  0xBAD\_C0FFEE  ${1}  ${1.0} |
| [float](https://docs.python.org/library/functions.html#float) | [Real](https://docs.python.org/library/numbers.html#numbers.Real) | double | [str](https://docs.python.org/library/functions.html#func-str), [Real](https://docs.python.org/library/numbers.html#numbers.Real) | Conversion is done using the [float](https://docs.python.org/library/functions.html#float) built-in.  Starting from RF 4.1, spaces and underscores can be used as visual separators for digit grouping purposes. | 3.14  2.9979e8  10 000.000 01  10\_000.000\_01 |
| [Decimal](https://docs.python.org/library/decimal.html#decimal.Decimal) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [int](https://docs.python.org/library/functions.html#int) , [float](https://docs.python.org/library/functions.html#float) | Conversion is done using the [Decimal](https://docs.python.org/library/decimal.html#decimal.Decimal) class. [Decimal](https://docs.python.org/library/decimal.html#decimal.Decimal) is recommended over [float](https://docs.python.org/library/functions.html#float) when decimal numbers need to be represented exactly.  Starting from RF 4.1, spaces and underscores can be used as visual separators for digit grouping purposes. | 3.14  10 000.000 01  10\_000.000\_01 |
| [str](https://docs.python.org/library/functions.html#func-str) |  | string, unicode | Any | All arguments are converted to Unicode strings. New in RF 4.0. |  |
| [bytes](https://docs.python.org/library/functions.html#func-bytes) | [ByteString](https://docs.python.org/library/collections.abc.html#collections.abc.ByteString) |  | [str](https://docs.python.org/library/functions.html#func-str), [bytearray](https://docs.python.org/library/functions.html" \l "func-bytearray) | Strings are converted to bytes so that each Unicode code point below 256 is directly mapped to a matching byte. Higher code points are not allowed. | good  hyvä (converted to hyv\xe4)  \x00 (the null byte) |
| [bytearray](https://docs.python.org/library/functions.html#func-bytearray) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [bytes](https://docs.python.org/library/functions.html#func-bytes) | Same conversion as with [bytes](https://docs.python.org/library/functions.html#func-bytes) but the result is a [bytearray](https://docs.python.org/library/functions.html" \l "func-bytearray). |  |
| [datetime](https://docs.python.org/library/datetime.html#datetime.datetime) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [int](https://docs.python.org/library/functions.html#int) , [float](https://docs.python.org/library/functions.html#float) | Strings are expected to be timestamps in [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) like format YYYY-MM-DD hh:mm:ss.mmmmmm, where any non-digit character can be used as a separator or separators can be omitted altogether. Additionally, only the date part is mandatory, all possibly missing time components are considered to be zeros.  Integers and floats are considered to represent seconds since the [Unix epoch](http://en.wikipedia.org/wiki/Unix_time). | 2022-02-09T16:39:43.632269  2022-02-09 16:39  2022-02-09  ${1644417583.632269} (Epoch time) |
| [date](https://docs.python.org/library/datetime.html#datetime.date) |  |  | [str](https://docs.python.org/library/functions.html#func-str) | Same string conversion as with [datetime](https://docs.python.org/library/datetime.html#datetime.datetime) but all time components are expected to be omitted or to be zeros. | 2018-09-12 |
| [timedelta](https://docs.python.org/library/datetime.html#datetime.timedelta) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [int](https://docs.python.org/library/functions.html#int) , [float](https://docs.python.org/library/functions.html#float) | Strings are expected to represent a time interval in one of the time formats Robot Framework supports: [time as number](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-as-number), [time as time string](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-as-time-string) or [time as "timer" string](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-as-timer-string). Integers and floats are considered to be seconds. | 42 (42 seconds)  1 minute 2 seconds  01:02 (same as above) |
| [Enum](https://docs.python.org/library/enum.html#enum.Enum) |  |  | [str](https://docs.python.org/library/functions.html#func-str) | The specified type must be an enumeration (a subclass of [Enum](https://docs.python.org/library/enum.html#enum.Enum) or [Flag](https://docs.python.org/library/enum.html#enum.Flag)) and given arguments must match its member names.  Starting from RF 3.2.2, matching member names is case-, space- and underscore-insensitive. | **class** **Direction**(Enum):  NORTH = auto()  NORTH\_WEST = auto()  NORTH (Direction.NORTH)  north west (Direction.NORTH\_WEST) |
| [IntEnum](https://docs.python.org/library/enum.html#enum.IntEnum) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [int](https://docs.python.org/library/functions.html#int) | The specified type must be an integer based enumeration (a subclass of [IntEnum](https://docs.python.org/library/enum.html" \l "enum.IntEnum) or [IntFlag](https://docs.python.org/library/enum.html" \l "enum.IntFlag)) and given arguments must match its member names or values.  Matching member names is case-, space- and and underscore-insensitive. Values can be given as actual integers and as strings that can be converted to integers.  Support for [IntEnum](https://docs.python.org/library/enum.html" \l "enum.IntEnum) and [IntFlag](https://docs.python.org/library/enum.html" \l "enum.IntFlag) is new in RF 4.1. | **class** **PowerState**(IntEnum):  OFF = 0  ON = 1  OFF (PowerState.OFF)  1 (PowerState.ON) |
| [None](https://docs.python.org/library/constants.html#None) |  | NoneType | [str](https://docs.python.org/library/functions.html#func-str) | String NONE (case-insensitive) is converted to the Python None object. Other values cause an error. | None |
| [list](https://docs.python.org/library/stdtypes.html#list) | [Sequence](https://docs.python.org/library/collections.abc.html#collections.abc.Sequence) |  | [str](https://docs.python.org/library/functions.html#func-str), [Sequence](https://docs.python.org/library/collections.abc.html#collections.abc.Sequence) | Strings must be Python list literals. They are converted to actual lists using the [ast.literal\_eval](https://docs.python.org/library/ast.html" \l "ast.literal_eval) function. They can contain any values ast.literal\_eval supports, including lists and other containers.  If the used type hint is [list](https://docs.python.org/library/stdtypes.html#list) (e.g. arg: list), sequences that are not lists are converted to lists. If the type hint is generic [Sequence](https://docs.python.org/library/collections.abc.html#collections.abc.Sequence), sequences are used without conversion. | ['one', 'two']  [('one', 1), ('two', 2)] |
| [tuple](https://docs.python.org/library/stdtypes.html#tuple) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [Sequence](https://docs.python.org/library/collections.abc.html#collections.abc.Sequence) | Same as [list](https://docs.python.org/library/stdtypes.html#list), but string arguments must tuple literals. | ('one', 'two') |
| [dict](https://docs.python.org/library/stdtypes.html#dict) | [Mapping](https://docs.python.org/library/collections.abc.html#collections.abc.Mapping) | dictionary, map | [str](https://docs.python.org/library/functions.html#func-str), [Mapping](https://docs.python.org/library/collections.abc.html#collections.abc.Mapping) | Same as [list](https://docs.python.org/library/stdtypes.html#list), but string arguments must be dictionary literals. | {'a': 1, 'b': 2}  {'key': 1, 'nested': {'key': 2}} |
| [set](https://docs.python.org/library/stdtypes.html#set) | [Set](https://docs.python.org/library/collections.abc.html#collections.abc.Set) |  | [str](https://docs.python.org/library/functions.html#func-str), [Container](https://docs.python.org/library/collections.abc.html#collections.abc.Container) | Same as [list](https://docs.python.org/library/stdtypes.html#list), but string arguments must be set literals or set() to create an empty set. | {1, 2, 3, 42}  set() |
| [frozenset](https://docs.python.org/library/stdtypes.html#frozenset) |  |  | [str](https://docs.python.org/library/functions.html#func-str), [Container](https://docs.python.org/library/collections.abc.html#collections.abc.Container) | Same conversion as with [set](https://docs.python.org/library/stdtypes.html#set), but the result is a [frozenset](https://docs.python.org/library/stdtypes.html" \l "frozenset). |  |

**Note**

Starting from Robot Framework 5.0, types that are automatically converted are automatically shown in [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) outputs.

**Note**

Prior to Robot Framework 4.0, most types supported converting string NONE (case-insensitively) to Python None. That support has been removed and None conversion is only done if an argument has None as an explicit type or as a default value.

**Specifying multiple possible types**

Starting from Robot Framework 4.0, it is possible to specify that an argument has multiple possible types. In this situation argument conversion is attempted based on each type and the whole conversion fails if none of these conversions succeed.

When using function annotations, the natural syntax to specify that argument has multiple possible types is using [Union](https://docs.python.org/3/library/typing.html#typing.Union):

**from** **typing** **import** Union

**def** example(length: Union[int, float], padding: Union[int, str, **None**] = **None**):

*# ...*

When using Python 3.10 or newer, it is possible to use the native type1 | type2 syntax instead:

**def** example(length: int | float, padding: int | str | **None** = **None**):

*# ...*

An alternative is specifying types as a tuple. It is not recommended with annotations, because that syntax is not supported by other tools, but it works well with the @keyword decorator:

**from** **robot.api.deco** **import** keyword

@keyword(types={'length': (int, float), 'padding': (int, str, **None**)})

**def** example(length, padding=**None**):

*# ...*

With the above examples the length argument would first be converted to an integer and if that fails then to a float. The padding would be first converted to an integer, then to a string, and finally to None.

If the given argument has one of the accepted types, then no conversion is done and the argument is used as-is. For example, if the length argument gets value 1.5 as a float, it would not be converted to an integer. Notice that using non-string values like floats as an argument requires using variables as these examples giving different values to the length argument demonstrate:

**\*\*\* Test Cases \*\*\***

**Conversion**

Example 10 *# Argument is a string. Converted to an integer.*

Example 1.5 *# Argument is a string. Converted to a float.*

Example ${10} *# Argument is an integer. Accepted as-is.*

Example ${1.5} *# Argument is a float. Accepted as-is.*

If one of the accepted types is string, then no conversion is done if the given argument is a string. As the following examples giving different values to the padding argument demonstrate, also in these cases passing other types is possible using variables:

**\*\*\* Test Cases \*\*\***

**Conversion**

Example 1 big *# Argument is a string. Accepted as-is.*

Example 1 10 *# Argument is a string. Accepted as-is.*

Example 1 ${10} *# Argument is an integer. Accepted as-is.*

Example 1 ${None} *# Argument is `None`. Accepted as-is.*

Example 1 ${1.5} *# Argument is a float. Converted to an integer.*

If the given argument does not have any of the accepted types, conversion is attempted in the order types are specified. If any conversion succeeds, the resulting value is used without attempting remaining conversions. If no individual conversion succeeds, the whole conversion fails.

If a specified type is not recognized by Robot Framework, then the original value is used as-is. For example, with this keyword conversion would first be attempted to an integer but if that fails the keyword would get the original given argument:

**def** example(argument: Union[int, MyCustomType]):

*# ...*

**Note**

In Robot Framework 4.0 argument conversion was done always, regardless of the type of the given argument. It caused [various](https://github.com/robotframework/robotframework/issues/3897) [problems](https://github.com/robotframework/robotframework/issues/3908) and was changed in Robot Framework 4.0.1.

**Custom argument converters**

In addition to doing argument conversion automatically as explained in the previous sections, Robot Framework supports custom argument conversion. This functionality has two main use cases:

* Overriding the standard argument converters provided by the framework.
* Adding argument conversion for custom types and for other types not supported out-of-the-box.

Argument converters are functions or other callables that get arguments used in data and convert them to desired format before arguments are passed to keywords. Converters are registered for libraries by setting ROBOT\_LIBRARY\_CONVERTERS attribute (case-sensitive) to a dictionary mapping desired types to converts. When implementing a library as a module, this attribute must be set on the module level, and with class based libraries it must be a class attribute. With libraries implemented as classes, it is also possible to use the converters argument with the [@library decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-decorator). Both of these approaches are illustrated by examples in the following sections.

**Note**

Custom argument converters are new in Robot Framework 5.0.

**Overriding default converters**

Let's assume we wanted to create a keyword that accepts [date](https://docs.python.org/library/datetime.html#datetime.date) objects for users in Finland where the commonly used date format is dd.mm.yyyy. The usage could look something like this:

**\*\*\* Test Cases \*\*\***

**Example**

Keyword 25.1.2022

[Automatic argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) supports dates, but it expects them to be in yyyy-mm-dd format so it will not work. A solution is creating a custom converter and registering it to handle [date](https://docs.python.org/library/datetime.html#datetime.date) conversion:

**from** **datetime** **import** date

*# Converter function.*

**def** parse\_fi\_date(value):

day, month, year = value.split('.')

**return** date(int(year), int(month), int(day))

*# Register converter function for the specified type.*

ROBOT\_LIBRARY\_CONVERTERS = {date: parse\_fi\_date}

*# Keyword using custom converter. Converter is got based on argument type.*

**def** keyword(arg: date):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

**Conversion errors**

If we try using the above keyword with invalid argument like invalid, it fails with this error:

ValueError: Argument 'arg' got value 'invalid' that cannot be converted to date: not enough values to unpack (expected 3, got 1)

This error is not too informative and does not tell anything about the expected format. Robot Framework cannot provide more information automatically, but the converter itself can be enhanced to validate the input. If the input is invalid, the converter should raise a ValueError with an appropriate message. In this particular case there would be several ways to validate the input, but using [regular expressions](https://en.wikipedia.org/wiki/Regular_expression) makes it possible to validate both that the input has dots (.) in correct places and that date parts contain correct amount of digits:

**from** **datetime** **import** date

**import** **re**

**def** parse\_fi\_date(value):

*# Validate input using regular expression and raise ValueError if not valid.*

match = re.match(r'(\d{1,2})\.(\d{1,2})\.(\d**{4}**)$', value)

**if** **not** match:

**raise** **ValueError**(f"Expected date in format 'dd.mm.yyyy', got '**{**value**}**'.")

day, month, year = match.groups()

**return** date(int(year), int(month), int(day))

ROBOT\_LIBRARY\_CONVERTERS = {date: parse\_fi\_date}

**def** keyword(arg: date):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

With the above converter code, using the keyword with argument invalid fails with a lot more helpful error message:

ValueError: Argument 'arg' got value 'invalid' that cannot be converted to date: Expected date in format 'dd.mm.yyyy', got 'invalid'.

**Restricting value types**

By default Robot Framework tries to use converters with all given arguments regardless their type. This means that if the earlier example keyword would be used with a variable containing something else than a string, conversion code would fail in the re.match call. For example, trying to use it with argument ${42} would fail like this:

ValueError: Argument 'arg' got value '42' (integer) that cannot be converted to date: TypeError: expected string or bytes-like object

This error situation could naturally handled in the converter code by checking the value type, but if the converter only accepts certain types, it is typically easier to just restrict the value to that type. Doing it requires only adding appropriate type hint to the converter:

**def** parse\_fi\_date(value: str):

*# ...*

Notice that this type hint *is not* used for converting the value before calling the converter, it is used for strictly restricting which types can be used. With the above addition calling the keyword with ${42} would fail like this:

ValueError: Argument 'arg' got value '42' (integer) that cannot be converted to date.

If the converter can accept multiple types, it is possible to specify types as a [Union](https://docs.python.org/3/library/typing.html#typing.Union). For example, if we wanted to enhance our keyword to accept also integers so that they would be considered seconds since the [Unix epoch](http://en.wikipedia.org/wiki/Unix_time), we could change the converter like this:

**from** **datetime** **import** date

**import** **re**

**from** **typing** **import** Union

*# Accept both strings and integers.*

**def** parse\_fi\_date(value: Union[str, int]):

*# Integers are converted separately.*

**if** isinstance(value, int):

**return** date.fromtimestamp(value)

match = re.match(r'(\d{1,2})\.(\d{1,2})\.(\d**{4}**)$', value)

**if** **not** match:

**raise** **ValueError**(f"Expected date in format 'dd.mm.yyyy', got '**{**value**}**'.")

day, month, year = match.groups()

**return** date(int(year), int(month), int(day))

ROBOT\_LIBRARY\_CONVERTERS = {date: parse\_fi\_date}

**def** keyword(arg: date):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

**Converting custom types**

A problem with the earlier example is that [date](https://docs.python.org/library/datetime.html#datetime.date) objects could only be given in dd.mm.yyyy format. It would not work if there was a need to support dates in different formats like in this example:

**\*\*\* Test Cases \*\*\***

**Example**

Finnish 25.1.2022

US 1/25/2022

ISO 8601 2022-01-22

A solution to this problem is creating custom types instead of overriding the default [date](https://docs.python.org/library/datetime.html#datetime.date) conversion:

**from** **datetime** **import** date

**import** **re**

**from** **typing** **import** Union

**from** **robot.api.deco** **import** keyword, library

*# Custom type. Extends an existing type but that is not required.*

**class** **FiDate**(date):

*# Converter function implemented as a classmethod. It could be a normal*

*# function as well, but this way all code is in the same class.*

@classmethod

**def** from\_string(cls, value: str):

match = re.match(r'(\d{1,2})\.(\d{1,2})\.(\d**{4}**)$', value)

**if** **not** match:

**raise** **ValueError**(f"Expected date in format 'dd.mm.yyyy', got '**{**value**}**'.")

day, month, year = match.groups()

**return** cls(int(year), int(month), int(day))

*# Another custom type.*

**class** **UsDate**(date):

@classmethod

**def** from\_string(cls, value: str):

match = re.match(r'(\d{1,2})/(\d{1,2})/(\d**{4}**)$', value)

**if** **not** match:

**raise** **ValueError**(f"Expected date in format 'mm/dd/yyyy', got '**{**value**}**'.")

month, day, year = match.groups()

**return** cls(int(year), int(month), int(day))

*# Register converters using '@library' decorator.*

@library(converters={FiDate: FiDate.from\_string, UsDate: UsDate.from\_string})

**class** **Library**:

*# Uses custom converter supporting 'dd.mm.yyyy' format.*

@keyword

**def** finnish(self, arg: FiDate):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

*# Uses custom converter supporting 'mm/dd/yyyy' format.*

@keyword

**def** us(self, arg: UsDate):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

*# Uses IS0-8601 compatible default conversion.*

@keyword

**def** iso\_8601(self, arg: date):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

*# Accepts date in different formats.*

@keyword

**def** any(self, arg: Union[FiDate, UsDate, date]):

print(f'year: **{**arg.year**}**, month: **{**arg.month**}**, day: **{**arg.day**}**')

**Converter documentation**

Information about converters is added to outputs produced by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) automatically. This information includes the name of the type, accepted values (if specified using type hints) and documentation. Type information is automatically linked to all keywords using these types.

Used documentation is got from the converter function by default. If it does not have any documentation, documentation is got from the type. Both of these approaches to add documentation to converters in the previous example thus produce the same result:

**class** **FiDate**(date):

@classmethod

**def** from\_string(cls, value: str):

*"""Date in ``dd.mm.yyyy`` format."""*

*# ...*

**class** **UsDate**(date):

*"""Date in ``mm/dd/yyyy`` format."""*

@classmethod

**def** from\_string(cls, value: str):

*# ...*

Adding documentation is in general recommended to provide users more information about conversion. It is especially important to document converter functions registered for existing types, because their own documentation is likely not very useful in this context.

[**@keyword decorator**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-526)

Although Robot Framework gets lot of information about keywords automatically, such as their names and arguments, there are sometimes needs to configure this information further. This is typically easiest done by using the robot.api.deco.keyword decorator. It has several useful usages that are explained thoroughly elsewhere and only listened here as a reference:

* Exposing methods and functions as keywords when the [automatic keyword discovery](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-public-methods-becoming-keywords) has been disabled by using the [@library decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-decorator) or otherwise.
* Setting a [custom name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name) to a keyword. This is especially useful when using the [embedded argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-names).
* Setting [keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags).
* Setting [type information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator) to enable automatic argument type conversion. Supports also disabling the argument conversion altogether.
* [Marking methods to expose as keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#marking-methods-to-expose-as-keywords) when using the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) or the [hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api).

[**@not\_keyword decorator**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-527)

The robot.api.deco.not\_keyword decorator can be used for [disabling functions or methods becoming keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-not-keyword-decorator).

[**Using custom decorators**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-528)

When implementing keywords, it is sometimes useful to modify them with [Python decorators](https://realpython.com/primer-on-python-decorators/). However, decorators often modify function signatures and can thus confuse Robot Framework's introspection when determining which arguments keywords accept. This is especially problematic when creating library documentation with [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) and when using external tools like [RIDE](https://github.com/robotframework/RIDE). The easiest way to avoid this problem is decorating the decorator itself using [functools.wraps](https://docs.python.org/library/functools.html" \l "functools.wraps). Other solutions include using external modules like [decorator](https://pypi.org/project/decorator/) and [wrapt](https://wrapt.readthedocs.io/) that allow creating fully signature-preserving decorators.

**Note**

Support for "unwrapping" decorators decorated with functools.wraps is a new feature in Robot Framework 3.2.

[**Embedding arguments into keyword names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-529)

Library keywords can also accept arguments which are passed using the [embedded argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#embedding-arguments-into-keyword-name). The [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-decorator) can be used to create a [custom keyword name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name) for the keyword which includes the desired syntax.

**from** **robot.api.deco** **import** keyword

@keyword('Add ${quantity:\d+} copies of $**{item}** to cart')

**def** add\_copies\_to\_cart(quantity, item):

*# ...*

**\*\*\* Test Cases \*\*\***

**My Test**

Add 7 copies of coffee to cart

By default arguments are passed to implementing keywords as strings, but automatic [argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-conversion) works if type information is specified somehow. It is convenient to use [function annotations](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations), and alternatively it is possible to pass types to the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator). This example uses annotations:

@keyword('Add ${quantity:\d+} copies of $**{item}** to cart')

**def** add\_copies\_to\_cart(quantity: int, item):

*# ...*

**Note**

Automatic type conversion is new in Robot Framework 3.1.

[**4.1.4   Communicating with Robot Framework**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-530)

After a method implementing a keyword is called, it can use any mechanism to communicate with the system under test. It can then also send messages to Robot Framework's log file, return information that can be saved to variables and, most importantly, report if the keyword passed or not.

[**Reporting keyword status**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-531)

Reporting keyword status is done simply using exceptions. If an executed method raises an exception, the keyword status is FAIL, and if it returns normally, the status is PASS.

Normal execution failures and errors can be reported using the standard exceptions such as AssertionError, ValueError and RuntimeError. There are, however, some special cases explained in the subsequent sections where special exceptions are needed.

**Error messages**

The error message shown in logs, reports and the console is created from the exception type and its message. With generic exceptions (for example, AssertionError, Exception, and RuntimeError), only the exception message is used, and with others, the message is created in the format ExceptionType: Actual message.

It is possible to avoid adding the exception type as a prefix to failure message also with non generic exceptions. This is done by adding a special ROBOT\_SUPPRESS\_NAME attribute with value True to your exception.

Python:

**class** **MyError**(**RuntimeError**):

ROBOT\_SUPPRESS\_NAME = **True**

In all cases, it is important for the users that the exception message is as informative as possible.

**HTML in error messages**

It is also possible to have HTML formatted error messages by starting the message with text \*HTML\*:

**raise** **AssertionError**("\*HTML\* <a href='robotframework.org'>Robot Framework</a> rulez!!")

This method can be used both when raising an exception in a library, like in the example above, and [when users provide an error message in the test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#failures).

**Cutting long messages automatically**

If the error message is longer than 40 lines, it will be automatically cut from the middle to prevent reports from getting too long and difficult to read. The full error message is always shown in the log message of the failed keyword.

**Tracebacks**

The traceback of the exception is also logged using DEBUG [log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). These messages are not visible in log files by default because they are very rarely interesting for normal users. When developing libraries, it is often a good idea to run tests using --loglevel DEBUG.

[**Exceptions provided by Robot Framework**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-532)

Robot Framework provides some exceptions that libraries can use for reporting failures and other events. These exceptions are exposed via the [robot.api](https://robot-framework.readthedocs.io/en/master/autodoc/robot.api.html) package and contain the following:

Failure

Report failed validation. There is no practical difference in using this exception compared to using the standard AssertionError. The main benefit of using this exception is that its name is consistent with other provided exceptions.

Error

Report error in execution. Failures related to the system not behaving as expected should typically be reported using the Failure exception or the standard AssertionError. This exception can be used, for example, if the keyword is used incorrectly. There is no practical difference, other than consistent naming with other provided exceptions, compared to using this exception and the standard RuntimeError.

ContinuableFailure

Report failed validation but allow continuing execution. See the [Continuable failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continuable-failures) section below for more information.

SkipExecution

Mark the executed test or task [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped). See the [Skipping tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipping-tests) section below for more information.

FatalError

Report error that stops the whole execution. See the [Stopping test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution) section below for more information.

**Note**

All these exceptions are new in Robot Framework 4.0. Other features than skipping tests, which is also new in Robot Framework 4.0, are available by other means in earlier versions.

[**Continuable failures**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-533)

It is possible to [continue test execution even when there are failures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure). The easiest way to do that is using the [provided](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#exceptions-provided-by-robot-framework) robot.api.ContinuableFailure exception:

**from** **robot.api** **import** ContinuableFailure

**def** example\_keyword():

**if** something\_is\_wrong():

**raise** ContinuableFailure('Something is wrong but execution can continue.')

...

An alternative is creating a custom exception that has a special ROBOT\_CONTINUE\_ON\_FAILURE attribute set to a True value. This is demonstrated by the example below.

**class** **MyContinuableError**(**RuntimeError**):

ROBOT\_CONTINUE\_ON\_FAILURE = **True**

[**Skipping tests**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-534)

It is possible to [skip](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skip) tests with a library keyword. The easiest way to do that is using the [provided](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#exceptions-provided-by-robot-framework) robot.api.SkipExecution exception:

**from** **robot.api** **import** SkipExecution

**def** example\_keyword():

**if** test\_should\_be\_skipped():

**raise** SkipExecution('Cannot proceed, skipping test.')

...

An alternative is creating a custom exception that has a special ROBOT\_SKIP\_EXECUTION attribute set to a True value. This is demonstrated by the example below.

**class** **MySkippingError**(**RuntimeError**):

ROBOT\_SKIP\_EXECUTION = **True**

[**Stopping test execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-535)

It is possible to fail a test case so that [the whole test execution is stopped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution-gracefully). The easiest way to accomplish this is using the [provided](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#exceptions-provided-by-robot-framework) robot.api.FatalError exception:

**from** **robot.api** **import** FatalError

**def** example\_keyword():

**if** system\_is\_not\_running():

**raise** FatalError('System is not running!')

...

In addition to using the robot.api.FatalError exception, it is possible create a custom exception that has a special ROBOT\_EXIT\_ON\_FAILURE attribute set to a True value. This is illustrated by the example below.

**class** **MyFatalError**(**RuntimeError**):

ROBOT\_EXIT\_ON\_FAILURE = **True**

[**Logging information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-536)

Exception messages are not the only way to give information to the users. In addition to them, methods can also send messages to [log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) simply by writing to the standard output stream (stdout) or to the standard error stream (stderr), and they can even use different [log levels](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). Another, and often better, logging possibility is using the [programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis).

By default, everything written by a method into the standard output is written to the log file as a single entry with the log level INFO. Messages written into the standard error are handled similarly otherwise, but they are echoed back to the original stderr after the keyword execution has finished. It is thus possible to use the stderr if you need some messages to be visible on the console where tests are executed.

**Using log levels**

To use other log levels than INFO, or to create several messages, specify the log level explicitly by embedding the level into the message in the format \*LEVEL\* Actual log message, where \*LEVEL\* must be in the beginning of a line and LEVEL is one of the available logging levels TRACE, DEBUG, INFO, WARN, ERROR and HTML.

**Errors and warnings**

Messages with ERROR or WARN level are automatically written to the console and a separate [Test Execution Errors section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) in the log files. This makes these messages more visible than others and allows using them for reporting important but non-critical problems to users.

**Logging HTML**

Everything normally logged by the library will be converted into a format that can be safely represented as HTML. For example, <b>foo</b> will be displayed in the log exactly like that and not as **foo**. If libraries want to use formatting, links, display images and so on, they can use a special pseudo log level HTML. Robot Framework will write these messages directly into the log with the INFO level, so they can use any HTML syntax they want. Notice that this feature needs to be used with care, because, for example, one badly placed </table> tag can ruin the log file quite badly.

When using the [public logging API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#public-logging-api), various logging methods have optional html attribute that can be set to True to enable logging in HTML format.

**Timestamps**

By default messages logged via the standard output or error streams get their timestamps when the executed keyword ends. This means that the timestamps are not accurate and debugging problems especially with longer running keywords can be problematic.

Keywords have a possibility to add an accurate timestamp to the messages they log if there is a need. The timestamp must be given as milliseconds since the [Unix epoch](http://en.wikipedia.org/wiki/Unix_time) and it must be placed after the [log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-log-levels) separated from it with a colon:

\*INFO:1308435758660\* Message with timestamp

\*HTML:1308435758661\* <b>HTML</b> message with timestamp

As illustrated by the examples below, adding the timestamp is easy. It is, however, even easier to get accurate timestamps using the [programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis). A big benefit of adding timestamps explicitly is that this approach works also with the [remote library interface](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library-interface).

**import** **time**

**def** example\_keyword():

print('\*INFO:**%d**\* Message with timestamp' % (time.time()\*1000))

**Logging to console**

If libraries need to write something to the console they have several options. As already discussed, warnings and all messages written to the standard error stream are written both to the log file and to the console. Both of these options have a limitation that the messages end up to the console only after the currently executing keyword finishes.

Another option is writing messages to sys.\_\_stdout\_\_ or sys.\_\_stderr\_\_. When using this approach, messages are written to the console immediately and are not written to the log file at all:

**import** **sys**

**def** my\_keyword(arg):

sys.\_\_stdout\_\_.write('Got arg **%s\n**' % arg)

The final option is using the [public logging API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#public-logging-api):

**from** **robot.api** **import** logger

**def** log\_to\_console(arg):

logger.console('Got arg **%s**' % arg)

**def** log\_to\_console\_and\_log\_file(arg):

logger.info('Got arg **%s**' % arg, also\_console=**True**)

**Logging example**

In most cases, the INFO level is adequate. The levels below it, DEBUG and TRACE, are useful for writing debug information. These messages are normally not shown, but they can facilitate debugging possible problems in the library itself. The WARN or ERROR level can be used to make messages more visible and HTML is useful if any kind of formatting is needed.

The following examples clarify how logging with different levels works.

print('Hello from a library.')

print('\*WARN\* Warning from a library.')

print('\*ERROR\* Something unexpected happen that may indicate a problem in the test.')

print('\*INFO\* Hello again!')

print('This will be part of the previous message.')

print('\*INFO\* This is a new message.')

print('\*INFO\* This is <b>normal text</b>.')

print('\*HTML\* This is <b>bold</b>.')

print('\*HTML\* <a href="http://robotframework.org">Robot Framework</a>')

|  |  |  |
| --- | --- | --- |
| 16:18:42.123 | INFO | Hello from a library. |
| 16:18:42.123 | WARN | Warning from a library. |
| 16:18:42.123 | ERROR | Something unexpected happen that may indicate a problem in the test. |
| 16:18:42.123 | INFO | Hello again! This will be part of the previous message. |
| 16:18:42.123 | INFO | This is a new message. |
| 16:18:42.123 | INFO | This is <b>normal text</b>. |
| 16:18:42.123 | INFO | This is **bold**. |
| 16:18:42.123 | INFO | [Robot Framework](http://robotframework.org/) |

[**Programmatic logging APIs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-537)

Programmatic APIs provide somewhat cleaner way to log information than using the standard output and error streams.

**Public logging API**

Robot Framework has a Python based logging API for writing messages to the log file and to the console. Test libraries can use this API like logger.info('My message') instead of logging through the standard output like print('\*INFO\* My message'). In addition to a programmatic interface being a lot cleaner to use, this API has a benefit that the log messages have accurate [timestamps](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamps).

The public logging API [is thoroughly documented](https://robot-framework.readthedocs.org/en/latest/autodoc/robot.api.html#module-robot.api.logger) as part of the API documentation at [https://robot-framework.readthedocs.org](https://robot-framework.readthedocs.org/). Below is a simple usage example:

**from** **robot.api** **import** logger

**def** my\_keyword(arg):

logger.debug('Got argument **%s**' % arg)

do\_something()

logger.info('<i>This</i> is a boring example', html=**True**)

logger.console('Hello, console!')

An obvious limitation is that test libraries using this logging API have a dependency to Robot Framework. If Robot Framework is not running, the messages are redirected automatically to Python's standard [logging](http://docs.python.org/library/logging.html) module.

**Using Python's standard logging module**

In addition to the new [public logging API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#public-logging-api), Robot Framework offers a built-in support to Python's standard [logging](http://docs.python.org/library/logging.html) module. This works so that all messages that are received by the root logger of the module are automatically propagated to Robot Framework's log file. Also this API produces log messages with accurate [timestamps](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamps), but logging HTML messages or writing messages to the console are not supported. A big benefit, illustrated also by the simple example below, is that using this logging API creates no dependency to Robot Framework.

**import** **logging**

**def** my\_keyword(arg):

logging.debug('Got argument **%s**' % arg)

do\_something()

logging.info('This is a boring example')

The logging module has slightly different log levels than Robot Framework. Its levels DEBUG, INFO, WARNING and ERROR are mapped directly to the matching Robot Framework log levels, and CRITICAL is mapped to ERROR. Custom log levels are mapped to the closest standard level smaller than the custom level. For example, a level between INFO and WARNING is mapped to Robot Framework's INFO level.

[**Logging during library initialization**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-538)

Libraries can also log during the test library import and initialization. These messages do not appear in the [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) like the normal log messages, but are instead written to the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log). This allows logging any kind of useful debug information about the library initialization. Messages logged using the WARN or ERROR levels are also visible in the [test execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) section in the log file.

Logging during the import and initialization is possible both using the [standard output and error streams](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-information) and the [programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis). Both of these are demonstrated below.

Library logging using the logging API during import:

**from** **robot.api** **import** logger

logger.debug("Importing library")

**def** keyword():

*# ...*

**Note**

If you log something during initialization, i.e. in Python \_\_init\_\_, the messages may be logged multiple times depending on the [library scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope).

[**Returning values**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-539)

The final way for keywords to communicate back to the core framework is returning information retrieved from the system under test or generated by some other means. The returned values can be [assigned to variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-values-from-keywords) in the test data and then used as inputs for other keywords, even from different test libraries.

Values are returned using the return statement in methods. Normally, one value is assigned into one [scalar variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variables), as illustrated in the example below. This example also illustrates that it is possible to return any objects and to use [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) to access object attributes.

**from** **mymodule** **import** MyObject

**def** return\_string():

**return** "Hello, world!"

**def** return\_object(name):

**return** MyObject(name)

**\*\*\* Test Cases \*\*\***

**Returning one value**

${string} = Return String

Should Be Equal ${string} Hello, world!

${object} = Return Object Robot

Should Be Equal ${object.name} Robot

Keywords can also return values so that they can be assigned into several [scalar variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#scalar-variables) at once, into [a list variable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#list-variables), or into scalar variables and a list variable. All these usages require that returned values are lists or list-like objects.

**def** return\_two\_values():

**return** 'first value', 'second value'

**def** return\_multiple\_values():

**return** ['a', 'list', 'of', 'strings']

**\*\*\* Test Cases \*\*\***

**Returning multiple values**

${var1} ${var2} = Return Two Values

Should Be Equal ${var1} first value

Should Be Equal ${var2} second value

@{list} = Return Two Values

Should Be Equal @{list}[0] first value

Should Be Equal @{list}[1] second value

${s1} ${s2} @{li} = Return Multiple Values

Should Be Equal ${s1} ${s2} a list

Should Be Equal @{li}[0] @{li}[1] of strings

[**Communication when using threads**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-540)

If a library uses threads, it should generally communicate with the framework only from the main thread. If a worker thread has, for example, a failure to report or something to log, it should pass the information first to the main thread, which can then use exceptions or other mechanisms explained in this section for communication with the framework.

This is especially important when threads are run on background while other keywords are running. Results of communicating with the framework in that case are undefined and can in the worst case cause a crash or a corrupted output file. If a keyword starts something on background, there should be another keyword that checks the status of the worker thread and reports gathered information accordingly.

Messages logged by non-main threads using the normal logging methods from [programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis) are silently ignored.

There is also a BackgroundLogger in separate [robotbackgroundlogger](https://github.com/robotframework/robotbackgroundlogger) project, with a similar API as the standard robot.api.logger. Normal logging methods will ignore messages from other than main thread, but the BackgroundLogger will save the background messages so that they can be later logged to Robot's log.

[**4.1.5   Distributing test libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-541)

[**Documenting libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-542)

A test library without documentation about what keywords it contains and what those keywords do is rather useless. To ease maintenance, it is highly recommended that library documentation is included in the source code and generated from it. Basically, that means using [docstrings](http://www.python.org/dev/peps/pep-0257) as in the example below.

**class** **MyLibrary**:

*"""This is an example library with some documentation."""*

**def** keyword\_with\_short\_documentation(self, argument):

*"""This keyword has only a short documentation"""*

**pass**

**def** keyword\_with\_longer\_documentation(self):

*"""First line of the documentation is here.*

*Longer documentation continues here and it can contain*

*multiple lines or paragraphs.*

*"""*

**pass**

Python has tools for creating an API documentation of a library documented as above. However, outputs from these tools can be slightly technical for some users. Another alternative is using Robot Framework's own documentation tool [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc). This tool can create a library documentation from libraries using the static library API, such as the ones above, but it also handles libraries using the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) and [hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api).

The first logical line of a keyword documentation, until the first empty line, is used for a special purpose and should contain a short overall description of the keyword. It is used as a *short documentation* by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) (for example, as a tool tip) and also shown in the [test logs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file).

By default documentation is considered to follow Robot Framework's [documentation formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) rules. This simple format allows often used styles like \*bold\* and \_italic\_, tables, lists, links, etc. It is possible to use also HTML, plain text and [reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText) formats. See the [Documentation format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-format) section for information how to set the format in the library source code and [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) chapter for more information about the formats in general.

**Note**

Prior to Robot Framework 3.1, the short documentation contained only the first physical line of the keyword documentation.

[**Testing libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-543)

Any non-trivial test library needs to be thoroughly tested to prevent bugs in them. Of course, this testing should be automated to make it easy to rerun tests when libraries are changed.

Python has excellent unit testing tools, and they suite very well for testing libraries. There are no major differences in using them for this purpose compared to using them for some other testing. The developers familiar with these tools do not need to learn anything new, and the developers not familiar with them should learn them anyway.

It is also easy to use Robot Framework itself for testing libraries and that way have actual end-to-end acceptance tests for them. There are plenty of useful keywords in the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library for this purpose. One worth mentioning specifically is *Run Keyword And Expect Error*, which is useful for testing that keywords report errors correctly.

Whether to use a unit- or acceptance-level testing approach depends on the context. If there is a need to simulate the actual system under test, it is often easier on the unit level. On the other hand, acceptance tests ensure that keywords do work through Robot Framework. If you cannot decide, of course it is possible to use both the approaches.

[**Packaging libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-544)

After a library is implemented, documented, and tested, it still needs to be distributed to the users. With simple libraries consisting of a single file, it is often enough to ask the users to copy that file somewhere and set the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path) accordingly. More complicated libraries should be packaged to make the installation easier.

Since libraries are normal programming code, they can be packaged using normal packaging tools. For information about packaging and distributing Python code see <https://packaging.python.org/>. When such a package is installed using [pip](https://pip.pypa.io/) or other tools, it is automatically in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path).

[**Deprecating keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-545)

Sometimes there is a need to replace existing keywords with new ones or remove them altogether. Just informing the users about the change may not always be enough, and it is more efficient to get warnings at runtime. To support that, Robot Framework has a capability to mark keywords *deprecated*. This makes it easier to find old keywords from the test data and remove or replace them.

Keywords can be deprecated by starting their documentation with text \*DEPRECATED, case-sensitive, and having a closing \* also on the first line of the documentation. For example, \*DEPRECATED\*, \*DEPRECATED.\*, and \*DEPRECATED in version 1.5.\* are all valid markers.

When a deprecated keyword is executed, a deprecation warning is logged and the warning is shown also in [the console and the Test Execution Errors section in log files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution). The deprecation warning starts with text Keyword '<name>' is deprecated. and has rest of the [short documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-libraries) after the deprecation marker, if any, afterwards. For example, if the following keyword is executed, there will be a warning like shown below in the log file.

**def** example\_keyword(argument):

*"""\*DEPRECATED!!\* Use keyword `Other Keyword` instead.*

*This keyword does something to given ``argument`` and returns results.*

*"""*

**return** do\_something(argument)

|  |  |  |
| --- | --- | --- |
| 20080911 16:00:22.650 | WARN | Keyword 'SomeLibrary.Example Keyword' is deprecated. Use keyword `Other Keyword` instead. |

This deprecation system works with most test libraries and also with [user keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-name-and-documentation).

[**4.1.6   Dynamic library API**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-546)

The dynamic API is in most ways similar to the static API. For example, reporting the keyword status, logging, and returning values works exactly the same way. Most importantly, there are no differences in importing dynamic libraries and using their keywords compared to other libraries. In other words, users do not need to know what APIs their libraries use.

Only differences between static and dynamic libraries are how Robot Framework discovers what keywords a library implements, what arguments and documentation these keywords have, and how the keywords are actually executed. With the static API, all this is done using reflection, but dynamic libraries have special methods that are used for these purposes.

One of the benefits of the dynamic API is that you have more flexibility in organizing your library. With the static API, you must have all keywords in one class or module, whereas with the dynamic API, you can, for example, implement each keyword as a separate class. This use case is not so important with Python, though, because its dynamic capabilities and multi-inheritance already give plenty of flexibility, and there is also possibility to use the [hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api).

Another major use case for the dynamic API is implementing a library so that it works as proxy for an actual library possibly running on some other process or even on another machine. This kind of a proxy library can be very thin, and because keyword names and all other information is got dynamically, there is no need to update the proxy when new keywords are added to the actual library.

This section explains how the dynamic API works between Robot Framework and dynamic libraries. It does not matter for Robot Framework how these libraries are actually implemented (for example, how calls to the run\_keyword method are mapped to a correct keyword implementation), and many different approaches are possible. Python users may also find the [PythonLibCore](https://github.com/robotframework/PythonLibCore) project useful.

[**Getting keyword names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-547)

Dynamic libraries tell what keywords they implement with the get\_keyword\_names method. This method cannot take any arguments, and it must return a list or array of strings containing the names of the keywords that the library implements.

If the returned keyword names contain several words, they can be returned separated with spaces or underscores, or in the camelCase format. For example, ['first keyword', 'second keyword'], ['first\_keyword', 'second\_keyword'], and ['firstKeyword', 'secondKeyword'] would all be mapped to keywords *First Keyword* and *Second Keyword*.

Dynamic libraries must always have this method. If it is missing, or if calling it fails for some reason, the library is considered a static library.

**Marking methods to expose as keywords**

If a dynamic library should contain both methods which are meant to be keywords and methods which are meant to be private helper methods, it may be wise to mark the keyword methods as such so it is easier to implement get\_keyword\_names. The robot.api.deco.keyword decorator allows an easy way to do this since it creates a [custom 'robot\_name' attribute](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name) on the decorated method. This allows generating the list of keywords just by checking for the robot\_name attribute on every method in the library during get\_keyword\_names.

**from** **robot.api.deco** **import** keyword

**class** **DynamicExample**:

**def** get\_keyword\_names(self):

*# Get all attributes and their values from the library.*

attributes = [(name, getattr(self, name)) **for** name **in** dir(self)]

*# Filter out attributes that do not have 'robot\_name' set.*

keywords = [(name, value) **for** name, value **in** attributes

**if** hasattr(value, 'robot\_name')]

*# Return value of 'robot\_name', if given, or the original 'name'.*

**return** [value.robot\_name **or** name **for** name, value **in** keywords]

**def** helper\_method(self):

*# ...*

@keyword

**def** keyword\_method(self):

*# ...*

[**Running keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-548)

Dynamic libraries have a special run\_keyword (alias runKeyword) method for executing their keywords. When a keyword from a dynamic library is used in the test data, Robot Framework uses the run\_keyword method to get it executed. This method takes two or three arguments. The first argument is a string containing the name of the keyword to be executed in the same format as returned by get\_keyword\_names. The second argument is a list of [positional arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments) given to the keyword in the test data, and the optional third argument is a dictionary containing [named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-arguments). If the third argument is missing, [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-dynamic-libraries) and [named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-dynamic-libraries) are not supported, and other named arguments are mapped to positional arguments.

**Note**

Prior to Robot Framework 3.1, normal named arguments were mapped to positional arguments regardless did run\_keyword accept two or three arguments. The third argument only got possible free named arguments.

After getting keyword name and arguments, the library can execute the keyword freely, but it must use the same mechanism to communicate with the framework as static libraries. This means using exceptions for reporting keyword status, logging by writing to the standard output or by using the provided logging APIs, and using the return statement in run\_keyword for returning something.

Every dynamic library must have both the get\_keyword\_names and run\_keyword methods but rest of the methods in the dynamic API are optional. The example below shows a working, albeit trivial, dynamic library.

**class** **DynamicExample**:

**def** get\_keyword\_names(self):

**return** ['first keyword', 'second keyword']

**def** run\_keyword(self, name, args, kwargs):

print("Running keyword '**%s**' with positional arguments **%s** and named arguments **%s**."

% (name, args, kwargs))

[**Getting keyword arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-549)

If a dynamic library only implements the get\_keyword\_names and run\_keyword methods, Robot Framework does not have any information about the arguments that the implemented keywords accept. For example, both *First Keyword* and *Second Keyword* in the example above could be used with any arguments. This is problematic, because most real keywords expect a certain number of keywords, and under these circumstances they would need to check the argument counts themselves.

Dynamic libraries can communicate what arguments their keywords expect by using the get\_keyword\_arguments (alias getKeywordArguments) method. This method gets the name of a keyword as an argument, and it must return a list of strings containing the arguments accepted by that keyword.

Similarly as other keywords, dynamic keywords can require any number of [positional arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-arguments), have [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values), accept [variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments), accept [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments) and have [named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments). The syntax how to represent all these different variables is derived from how they are specified in Python and explained in the following table.

| *Representing different arguments with get\_keyword\_arguments* | | |
| --- | --- | --- |
| **Argument type** | **How to represent** | **Examples** |
| No arguments | Empty list. | [] |
| One or more [positional argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-argument) | List of strings containing argument names. | ['argument'], ['arg1', 'arg2', 'arg3'] |
| [Default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#default-values) | Two ways how to represent the argument name and the default value:   * As a string where the name and the default are separated with =. * As a tuple with the name and the default as separate items. New in Robot Framework 3.2. | ['name=default'], ['a', 'b=1', 'c=2']  [('name', 'default')], ['a', ('b', 1), ('c', 2)] |
| [Variable number of arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-number-of-arguments) (varargs) | Argument after possible positional arguments and their defaults has \* prefix. | ['\*varargs'], ['argument', '\*rest'], ['a', 'b=42', '\*c'] |
| [Free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments) (kwargs) | Last arguments has \*\* prefix. Requires run\_keyword to [support free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-dynamic-libraries). | ['\*\*named'], ['a', 'b=42', '\*\*c'], ['\*varargs', '\*\*kwargs'] |
| [Named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments) | Arguments after varargs or a lone \* if there are no varargs. With or without defaults. Requires run\_keyword to [support named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments-with-dynamic-libraries). New in Robot Framework 3.1. | ['\*varargs', 'named'], ['\*', 'named'], `['\*', 'x', 'y=default'], ['a', '\*b', 'c', '\*\*d'] |

When the get\_keyword\_arguments is used, Robot Framework automatically calculates how many positional arguments the keyword requires and does it support free named arguments or not. If a keyword is used with invalid arguments, an error occurs and run\_keyword is not even called.

The actual argument names and default values that are returned are also important. They are needed for [named argument support](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax-with-dynamic-libraries) and the [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) tool needs them to be able to create a meaningful library documentation.

As explained in the above table, default values can be specified with argument names either as a string like 'name=default' or as a tuple like ('name', 'default'). The main problem with the former syntax is that all default values are considered strings whereas the latter syntax allows using all objects like ('inteter', 1) or ('boolean', True). When using other objects than strings, Robot Framework can do [automatic argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values) based on them.

For consistency reasons, also arguments that do not accept default values can be specified as one item tuples. For example, ['a', 'b=c', '\*d'] and [('a',), ('b', 'c'), ('\*d',)] are equivalent.

If get\_keyword\_arguments is missing or returns Python None for a certain keyword, that keyword gets an argument specification accepting all arguments. This automatic argument spec is either [\*varargs, \*\*kwargs] or [\*varargs], depending does run\_keyword [support free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-dynamic-libraries) or not.

**Note**

Support to specify arguments as tuples like ('name', 'default') is new in Robot Framework 3.2.

[**Getting keyword argument types**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-550)

Robot Framework 3.1 introduced support for automatic argument conversion and the dynamic library API supports that as well. The conversion logic works exactly like with [static libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-conversion), but how the type information is specified is naturally different.

With dynamic libraries types can be returned using the optional get\_keyword\_types method (alias getKeywordTypes). It can return types using a list or a dictionary exactly like types can be specified when using the [@keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator). Type information can be specified using actual types like int, but especially if a dynamic library gets this information from external systems, using strings like 'int' or 'integer' may be easier. See the [Supported conversions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) section for more information about supported types and how to specify them.

Robot Framework does automatic argument conversion also based on the [argument default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values). Earlier this did not work with the dynamic API because it was possible to specify arguments only as strings. As [discussed in the previous section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments), this was changed in Robot Framework 3.2 and nowadays default values returned like ('example', True) are automatically used for this purpose.

[**Getting keyword tags**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-551)

Dynamic libraries can report [keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags) by using the get\_keyword\_tags method (alias getKeywordTags). It gets a keyword name as an argument, and should return corresponding tags as a list of strings.

Alternatively it is possible to specify tags on the last row of the documentation returned by the get\_keyword\_documentation method discussed below. This requires starting the last row with Tags: and listing tags after it like Tags: first tag, second, third.

**Tip**

The get\_keyword\_tags method is guaranteed to be called before the get\_keyword\_documentation method. This makes it easy to embed tags into the documentation only if the get\_keyword\_tags method is not called.

[**Getting keyword documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-552)

If dynamic libraries want to provide keyword documentation, they can implement the get\_keyword\_documentation method (alias getKeywordDocumentation). It takes a keyword name as an argument and, as the method name implies, returns its documentation as a string.

The returned documentation is used similarly as the keyword documentation string with static libraries. The main use case is getting keywords' documentations into a library documentation generated by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc). Additionally, the first line of the documentation (until the first \n) is shown in test logs.

[**Getting general library documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-553)

The get\_keyword\_documentation method can also be used for specifying overall library documentation. This documentation is not used when tests are executed, but it can make the documentation generated by [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) much better.

Dynamic libraries can provide both general library documentation and documentation related to taking the library into use. The former is got by calling get\_keyword\_documentation with special value \_\_intro\_\_, and the latter is got using value \_\_init\_\_. How the documentation is presented is best tested with [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) in practice.

Dynamic libraries can also specify the general library documentation directly in the code as the docstring of the library class and its \_\_init\_\_ method. If a non-empty documentation is got both directly from the code and from the get\_keyword\_documentation method, the latter has precedence.

[**Getting keyword source information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-554)

The dynamic API masks the real implementation of keywords from Robot Framework and thus makes it impossible to see where keywords are implemented. This means that editors and other tools utilizing Robot Framework APIs cannot implement features such as go-to-definition. This problem can be solved by implementing yet another optional dynamic method named get\_keyword\_source (alias getKeywordSource) that returns the source information.

The return value from the get\_keyword\_source method must be a string or None if no source information is available. In the simple case it is enough to simply return an absolute path to the file implementing the keyword. If the line number where the keyword implementation starts is known, it can be embedded to the return value like path:lineno. Returning only the line number is possible like :lineno.

The source information of the library itself is got automatically from the imported library class the same way as with other library APIs. The library source path is used with all keywords that do not have their own source path defined.

**Note**

Returning source information for keywords is a new feature in Robot Framework 3.2.

[**Named argument syntax with dynamic libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-555)

Also the dynamic library API supports the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax). Using the syntax works based on the argument names and default values [got from the library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments) using the get\_keyword\_arguments method.

If the run\_keyword method accepts three arguments, the second argument gets all positional arguments as a list and the last arguments gets all named arguments as a mapping. If it accepts only two arguments, named arguments are mapped to positional arguments. In the latter case, if a keyword has multiple arguments with default values and only some of the latter ones are given, the framework fills the skipped optional arguments based on the default values returned by the get\_keyword\_arguments method.

Using the named argument syntax with dynamic libraries is illustrated by the following examples. All the examples use a keyword *Dynamic* that has an argument specification [a, b=d1, c=d2]. The comment on each row shows how run\_keyword would be called in these cases if it has two arguments (i.e. signature is name, args) and if it has three arguments (i.e. name, args, kwargs).

**\*\*\* Test Cases \*\*\*** *# args # args, kwargs*

**Positional only**

Dynamic x *# [x] # [x], {}*

Dynamic x y *# [x, y] # [x, y], {}*

Dynamic x y z *# [x, y, z] # [x, y, z], {}*

**Named only**

Dynamic a=x *# [x] # [], {a: x}*

Dynamic c=z a=x b=y *# [x, y, z] # [], {a: x, b: y, c: z}*

**Positional and named**

Dynamic x b=y *# [x, y] # [x], {b: y}*

Dynamic x y c=z *# [x, y, z] # [x, y], {c: z}*

Dynamic x b=y c=z *# [x, y, z] # [x], {y: b, c: z}*

**Intermediate missing**

Dynamic x c=z *# [x, d1, z] # [x], {c: z}*

**Note**

Prior to Robot Framework 3.1, all normal named arguments were mapped to positional arguments and the optional kwargs was only used with free named arguments. With the above examples run\_keyword was always called like it is nowadays called if it does not support kwargs.

[**Free named arguments with dynamic libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-556)

Dynamic libraries can also support [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments) (\*\*named). A mandatory precondition for this support is that the run\_keyword method [takes three arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-dynamic-keywords): the third one will get the free named arguments along with possible other named arguments. These arguments are passed to the keyword as a mapping.

What arguments a keyword accepts depends on what get\_keyword\_arguments [returns for it](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments). If the last argument starts with \*\*, that keyword is recognized to accept free named arguments.

Using the free named argument syntax with dynamic libraries is illustrated by the following examples. All the examples use a keyword *Dynamic* that has an argument specification [a=d1, b=d2, \*\*named]. The comment shows the arguments that the run\_keyword method is actually called with.

**\*\*\* Test Cases \*\*\*** *# args, kwargs*

**No arguments**

Dynamic *# [], {}*

**Positional only**

Dynamic x *# [x], {}*

Dynamic x y *# [x, y], {}*

**Free named only**

Dynamic x=1 *# [], {x: 1}*

Dynamic x=1 y=2 z=3 *# [], {x: 1, y: 2, z: 3}*

**Free named with positional**

Dynamic x y=2 *# [x], {y: 2}*

Dynamic x y=2 z=3 *# [x], {y: 2, z: 3}*

**Free named with normal named**

Dynamic a=1 x=1 *# [], {a: 1, x: 1}*

Dynamic b=2 x=1 a=1 *# [], {a: 1, b: 2, x: 1}*

**Note**

Prior to Robot Framework 3.1, normal named arguments were mapped to positional arguments but nowadays they are part of the kwargs along with the free named arguments.

[**Named-only arguments with dynamic libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-557)

Starting from Robot Framework 3.1, dynamic libraries can have [named-only arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-only-arguments). This requires that the run\_keyword method [takes three arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-dynamic-keywords): the third getting the named-only arguments along with the other named arguments.

In the [argument specification](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments) returned by the get\_keyword\_arguments method named-only arguments are specified after possible variable number of arguments (\*varargs) or a lone asterisk (\*) if the keyword does not accept varargs. Named-only arguments can have default values, and the order of arguments with and without default values does not matter.

Using the named-only argument syntax with dynamic libraries is illustrated by the following examples. All the examples use a keyword *Dynamic* that has been specified to have argument specification [positional=default, \*varargs, named, named2=default, \*\*free]. The comment shows the arguments that the run\_keyword method is actually called with.

**\*\*\* Test Cases \*\*\*** *# args, kwargs*

**Named-only only**

Dynamic named=value *# [], {named: value}*

Dynamic named=value named2=2 *# [], {named: value, named2: 2}*

**Named-only with positional and varargs**

Dynamic argument named=xxx *# [argument], {named: xxx}*

Dynamic a1 a2 named=3 *# [a1, a2], {named: 3}*

**Named-only with normal named**

Dynamic named=foo positional=bar *# [], {positional: bar, named: foo}*

**Named-only with free named**

Dynamic named=value foo=bar *# [], {named: value, foo=bar}*

Dynamic named2=2 third=3 named=1 *# [], {named: 1, named2: 2, third: 3}*

[**Summary**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-558)

All special methods in the dynamic API are listed in the table below. Method names are listed in the underscore format, but their camelCase aliases work exactly the same way.

| *All special methods in the dynamic API* | | |
| --- | --- | --- |
| **Name** | **Arguments** | **Purpose** |
| get\_keyword\_names |  | [Return names](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-dynamic-keyword-names) of the implemented keywords. |
| run\_keyword | name, arguments, kwargs | [Execute the specified keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#running-dynamic-keywords) with given arguments. kwargs is optional. |
| get\_keyword\_arguments | name | Return keywords' [argument specification](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments). Optional method. |
| get\_keyword\_types | name | Return keywords' [argument type information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-argument-types). Optional method. New in RF 3.1. |
| get\_keyword\_tags | name | Return keywords' [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-tags). Optional method. |
| get\_keyword\_documentation | name | Return keywords' and library's [documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-documentation). Optional method. |
| get\_keyword\_source | name | Return keywords' [source](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-source-information). Optional method. New in RF 3.2. |

A good example of using the dynamic API is Robot Framework's own [Remote library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-library).

[**4.1.7   Hybrid library API**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-559)

The hybrid library API is, as its name implies, a hybrid between the static API and the dynamic API. Just as with the dynamic API, it is possible to implement a library using the hybrid API only as a class.

[**Getting keyword names**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-560)

Keyword names are got in the exactly same way as with the dynamic API. In practice, the library needs to have the get\_keyword\_names or getKeywordNames method returning a list of keyword names that the library implements.

[**Running keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-561)

In the hybrid API, there is no run\_keyword method for executing keywords. Instead, Robot Framework uses reflection to find methods implementing keywords, similarly as with the static API. A library using the hybrid API can either have those methods implemented directly or, more importantly, it can handle them dynamically.

In Python, it is easy to handle missing methods dynamically with the \_\_getattr\_\_ method. This special method is probably familiar to most Python programmers and they can immediately understand the following example. Others may find it easier to consult [Python Reference Manual](http://docs.python.org/reference/datamodel.html#attribute-access) first.

**from** **somewhere** **import** external\_keyword

**class** **HybridExample**:

**def** get\_keyword\_names(self):

**return** ['my\_keyword', 'external\_keyword']

**def** my\_keyword(self, arg):

print("My Keyword called with '**%s**'" % arg)

**def** \_\_getattr\_\_(self, name):

**if** name == 'external\_keyword':

**return** external\_keyword

**raise** **AttributeError**("Non-existing attribute '**%s**'" % name)

Note that \_\_getattr\_\_ does not execute the actual keyword like run\_keyword does with the dynamic API. Instead, it only returns a callable object that is then executed by Robot Framework.

Another point to be noted is that Robot Framework uses the same names that are returned from get\_keyword\_names for finding the methods implementing them. Thus the names of the methods that are implemented in the class itself must be returned in the same format as they are defined. For example, the library above would not work correctly, if get\_keyword\_names returned My Keyword instead of my\_keyword.

[**Getting keyword arguments and documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-562)

When this API is used, Robot Framework uses reflection to find the methods implementing keywords, similarly as with the static API. After getting a reference to the method, it searches for arguments and documentation from it, in the same way as when using the static API. Thus there is no need for special methods for getting arguments and documentation like there is with the dynamic API.

[**Summary**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-563)

When implementing a test library, the hybrid API has the same dynamic capabilities as the actual dynamic API. A great benefit with it is that there is no need to have special methods for getting keyword arguments and documentation. It is also often practical that the only real dynamic keywords need to be handled in \_\_getattr\_\_ and others can be implemented directly in the main library class.

Because of the clear benefits and equal capabilities, the hybrid API is in most cases a better alternative than the dynamic API. One notable exception is implementing a library as a proxy for an actual library implementation elsewhere, because then the actual keyword must be executed elsewhere and the proxy can only pass forward the keyword name and arguments.

A good example of using the hybrid API is Robot Framework's own [Telnet](https://robotframework.org/robotframework/latest/libraries/Telnet.html) library.

[**4.1.8   Using Robot Framework's internal modules**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-564)

Test libraries can use Robot Framework's internal modules, for example, to get information about the executed tests and the settings that are used. This powerful mechanism to communicate with the framework should be used with care, though, because all Robot Framework's APIs are not meant to be used by externally and they might change radically between different framework versions.

[**Available APIs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-565)

[API documentation](http://robot-framework.readthedocs.org/) is hosted separately at the excellent [Read the Docs](http://readthedocs.org/) service. If you are unsure how to use certain API or is using them forward compatible, please send a question to [mailing list](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#mailing-lists).

[**Using BuiltIn library**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-566)

The safest API to use are methods implementing keywords in the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library. Changes to keywords are rare and they are always done so that old usage is first deprecated. One of the most useful methods is replace\_variables which allows accessing currently available variables. The following example demonstrates how to get ${OUTPUT\_DIR} which is one of the many handy [automatic variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatic-variables). It is also possible to set new variables from libraries using set\_test\_variable, set\_suite\_variable and set\_global\_variable.

**import** **os.path**

**from** **robot.libraries.BuiltIn** **import** BuiltIn

**def** do\_something(argument):

output = do\_something\_that\_creates\_a\_lot\_of\_output(argument)

outputdir = BuiltIn().replace\_variables('$**{OUTPUTDIR}**')

path = os.path.join(outputdir, 'results.txt')

f = open(path, 'w')

f.write(output)

f.close()

print('\*HTML\* Output written to <a href="results.txt">results.txt</a>')

The only catch with using methods from BuiltIn is that all run\_keyword method variants must be handled specially. Methods that use run\_keyword methods have to be registered as *run keywords* themselves using register\_run\_keyword method in BuiltIn module. This method's documentation explains why this needs to be done and obviously also how to do it.

[**4.1.9   Extending existing test libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-567)

This section explains different approaches how to add new functionality to existing test libraries and how to use them in your own libraries otherwise.

[**Modifying original source code**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-568)

If you have access to the source code of the library you want to extend, you can naturally modify the source code directly. The biggest problem of this approach is that it can be hard for you to update the original library without affecting your changes. For users it may also be confusing to use a library that has different functionality than the original one. Repackaging the library may also be a big extra task.

This approach works extremely well if the enhancements are generic and you plan to submit them back to the original developers. If your changes are applied to the original library, they are included in the future releases and all the problems discussed above are mitigated. If changes are non-generic, or you for some other reason cannot submit them back, the approaches explained in the subsequent sections probably work better.

[**Using inheritance**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-569)

Another straightforward way to extend an existing library is using inheritance. This is illustrated by the example below that adds new *Title Should Start With* keyword to the [SeleniumLibrary](https://github.com/robotframework/SeleniumLibrary).

**from** **SeleniumLibrary** **import** SeleniumLibrary

**class** **ExtendedSeleniumLibrary**(SeleniumLibrary):

**def** title\_should\_start\_with(self, expected):

title = self.get\_title()

**if** **not** title.startswith(expected):

**raise** **AssertionError**("Title '**%s**' did not start with '**%s**'"

% (title, expected))

A big difference with this approach compared to modifying the original library is that the new library has a different name than the original. A benefit is that you can easily tell that you are using a custom library, but a big problem is that you cannot easily use the new library with the original. First of all your new library will have same keywords as the original meaning that there is always [conflict](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-keywords-with-same-names). Another problem is that the libraries do not share their state.

This approach works well when you start to use a new library and want to add custom enhancements to it from the beginning. Otherwise other mechanisms explained in this section are probably better.

[**Using other libraries directly**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-570)

Because test libraries are technically just classes or modules, a simple way to use another library is importing it and using its methods. This approach works great when the methods are static and do not depend on the library state. This is illustrated by the earlier example that uses [Robot Framework's BuiltIn library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-robot-framework-s-internal-modules).

If the library has state, however, things may not work as you would hope. The library instance you use in your library will not be the same as the framework uses, and thus changes done by executed keywords are not visible to your library. The next section explains how to get an access to the same library instance that the framework uses.

[**Getting active library instance from Robot Framework**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-571)

[BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Get Library Instance* can be used to get the currently active library instance from the framework itself. The library instance returned by this keyword is the same as the framework itself uses, and thus there is no problem seeing the correct library state. Although this functionality is available as a keyword, it is typically used in test libraries directly by importing the *BuiltIn* library class [as discussed earlier](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-robot-framework-s-internal-modules). The following example illustrates how to implement the same *Title Should Start With* keyword as in the earlier example about [using inheritance](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-inheritance).

**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))

This approach is clearly better than importing the library directly and using it when the library has a state. The biggest benefit over inheritance is that you can use the original library normally and use the new library in addition to it when needed. That is demonstrated in the example below where the code from the previous examples is expected to be available in a new library *SeLibExtensions*.

**\*\*\* Settings \*\*\***

**Library** **SeleniumLibrary**

**Library** **SeLibExtensions**

**\*\*\* Test Cases \*\*\***

**Example**

Open Browser http://example *# SeleniumLibrary*

Title Should Start With Example *# SeLibExtensions*

[**Libraries using dynamic or hybrid API**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-572)

Test libraries that use the [dynamic](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) or [hybrid library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#hybrid-library-api) often have their own systems how to extend them. With these libraries you need to ask guidance from the library developers or consult the library documentation or source code.

[**4.2   Remote library interface**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-501)

The remote library interface provides means for having test libraries on different machines than where Robot Framework itself is running, and also for implementing libraries using other languages than the natively supported Python. For a test library, user remote libraries look pretty much the same as any other test library, and developing test libraries using the remote library interface is also very close to creating [normal test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries).

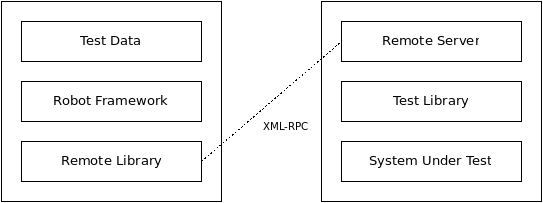
* [4.2.1   Introduction](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#introduction-3)
* [4.2.2   Putting Remote library to use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#putting-remote-library-to-use)
  + [Importing Remote library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-remote-library)
  + [Starting and stopping remote servers](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#starting-and-stopping-remote-servers)
* [4.2.3   Supported argument and return value types](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-argument-and-return-value-types)
* [4.2.4   Remote protocol](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-protocol)
  + [Required methods](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#required-methods)
  + [Using get\_keyword\_names and keyword specific getters](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-get-keyword-names-and-keyword-specific-getters)
  + [Using get\_library\_information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-get-library-information)
  + [Executing remote keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-remote-keywords)
  + [Different argument syntaxes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-argument-syntaxes)

[**4.2.1   Introduction**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-573)

There are two main reasons for using the remote library API:

* It is possible to have actual libraries on different machines than where Robot Framework is running. This allows interesting possibilities for distributed testing.
* Test libraries can be implemented using any language that supports [XML-RPC](http://www.xmlrpc.com/) protocol. There exists ready-made [generic remote servers](https://github.com/robotframework/RemoteInterface#available-remote-servers) for various languages like Python, Java, Ruby, .NET, and so on.

The remote library interface is provided by the Remote library that is one of the [standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#standard-libraries). This library does not have any keywords of its own, but it works as a proxy between the core framework and keywords implemented elsewhere. The Remote library interacts with actual library implementations through remote servers, and the Remote library and servers communicate using a simple [remote protocol](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#remote-protocol) on top of an XML-RPC channel. The high level architecture of all this is illustrated in the picture below:



*Robot Framework architecture with Remote library*

**Note**

The remote client uses Python's standard [XML-RPC module](https://docs.python.org/library/xmlrpc.client.html). It does not support custom XML-RPC extensions implemented by some XML-RPC servers.

[**4.2.2   Putting Remote library to use**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-574)

[**Importing Remote library**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-575)

The Remote library needs to know the address of the remote server but otherwise importing it and using keywords that it provides is no different to how other libraries are used. If you need to use the Remote library multiple times in a test suite, or just want to give it a more descriptive name, you can import it using the [WITH NAME syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name-to-test-library).

**\*\*\* Settings \*\*\***

**Library** **Remote** http://127.0.0.1:8270 WITH NAME Example1

**Library** **Remote** http://example.com:8080/ WITH NAME Example2

**Library** **Remote** http://10.0.0.2/example 1 minute WITH NAME Example3

The URL used by the first example above is also the default address that the Remote library uses if no address is given.

The last example above shows how to give a custom timeout to the Remote library as an optional second argument. The timeout is used when initially connecting to the server and if a connection accidentally closes. Timeout can be given in Robot Framework [time format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format) like 60s or 2 minutes 10 seconds. The default timeout is typically several minutes, but it depends on the operating system and its configuration. Notice that setting a timeout that is shorter than keyword execution time will interrupt the keyword.

**Note**

Port 8270 is the default port that remote servers are expected to use and it has been [registered by IANA](http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=8270) for this purpose. This port number was selected because 82 and 70 are the ASCII codes of letters R and F, respectively.

**Note**

When connecting to the local machine, it is recommended to use IP address 127.0.0.1 instead of machine name localhost. This avoids address resolution that can be extremely slow [at least on Windows](http://stackoverflow.com/questions/14504450/pythons-xmlrpc-extremely-slow-one-second-per-call).

**Note**

If the URI contains no path after the server address, the [XML-RPC module](https://docs.python.org/library/xmlrpc.client.html) used by the Remote library will use /RPC2 path by default. In practice using http://127.0.0.1:8270 is thus identical to using http://127.0.0.1:8270/RPC2. Depending on the remote server this may or may not be a problem. No extra path is appended if the address has a path even if the path is just /. For example, neither http://127.0.0.1:8270/ nor http://127.0.0.1:8270/my/path will be modified.

[**Starting and stopping remote servers**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-576)

Before the Remote library can be imported, the remote server providing the actual keywords must be started. If the server is started before launching the test execution, it is possible to use the normal *Library* setting like in the above example. Alternatively other keywords, for example from [Process](https://robotframework.org/robotframework/latest/libraries/Process.html) or [SSH](https://github.com/robotframework/SSHLibrary) libraries, can start the server up, but then you may need to use [Import Library keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-import-library-keyword) because the library is not available when the test execution starts.

How a remote server can be stopped depends on how it is implemented. Typically servers support the following methods:

* Regardless of the library used, remote servers should provide *Stop Remote Server* keyword that can be easily used by executed tests.
* Remote servers should have stop\_remote\_server method in their XML-RPC interface.
* Hitting Ctrl-C on the console where the server is running should stop the server.
* The server process can be terminated using tools provided by the operating system (e.g. kill).

**Note**

Servers may be configured so that users cannot stop it with *Stop Remote Server* keyword or stop\_remote\_server method.

[**4.2.3   Supported argument and return value types**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-577)

Because the XML-RPC protocol does not support all possible object types, the values transferred between the Remote library and remote servers must be converted to compatible types. This applies to the keyword arguments the Remote library passes to remote servers and to the return values servers give back to the Remote library.

Both the Remote library and the Python remote server handle Python values according to the following rules. Other remote servers should behave similarly.

* Strings, numbers and Boolean values are passed without modifications.
* Python None is converted to an empty string.
* All lists, tuples, and other iterable objects (except strings and dictionaries) are passed as lists so that their contents are converted recursively.
* Dictionaries and other mappings are passed as dicts so that their keys are converted to strings and values converted to supported types recursively.
* Returned dictionaries are converted to so called *dot-accessible dicts* that allow accessing keys as attributes using the [extended variable syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extended-variable-syntax) like ${result.key}. This works also with nested dictionaries like ${root.child.leaf}.
* Strings containing bytes in the ASCII range that cannot be represented in XML (e.g. the null byte) are sent as [Binary objects](http://docs.python.org/library/xmlrpc.client.html#binary-objects) that internally use XML-RPC base64 data type. Received Binary objects are automatically converted to byte strings.
* Other types are converted to strings.

[**4.2.4   Remote protocol**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-578)

This section explains the protocol that is used between the Remote library and remote servers. This information is mainly targeted for people who want to create new remote servers.

The remote protocol is implemented on top of [XML-RPC](http://www.xmlrpc.com/), which is a simple remote procedure call protocol using XML over HTTP. Most mainstream languages (Python, Java, C, Ruby, Perl, Javascript, PHP, ...) have a support for XML-RPC either built-in or as an extension.

The [Python remote server](https://github.com/robotframework/PythonRemoteServer) can be used as a reference implementation.

[**Required methods**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-579)

There are two possibilities how remote servers can provide information about the keywords they contain. They are briefly explained below and documented more thoroughly in the subsequent sections.

1. Remote servers can implement the same methods as the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) has. This means get\_keyword\_names method and optional get\_keyword\_arguments, get\_keyword\_types, get\_keyword\_tags and get\_keyword\_documentation methods. Notice that using "camel-case names" like getKeywordNames is not possible similarly as in the normal dynamic API.
2. Starting from Robot Framework 4.0, remote servers can have a single get\_library\_information method that returns all library and keyword information as a single dictionary. If a remote server has this method, the other getter methods like get\_keyword\_names are not used at all. This approach has the benefit that there is only one XML-RPC call to get information while the approach explained above requires several calls per keyword. With bigger libraries the difference can be significant.

Regardless how remote servers provide information about their keywords, they must have run\_keyword method that is used when keywords are executed. How the actual keywords are implemented is not relevant for the Remote library. Remote servers can either act as wrappers for the real test libraries, like the available [generic remote servers](https://github.com/robotframework/RemoteInterface#available-remote-servers) do, or they can implement keywords themselves.

Remote servers should additionally have stop\_remote\_server method in their public interface to ease stopping them. They should also automatically expose this method as *Stop Remote Server* keyword to allow using it in the test data regardless of the test library. Allowing users to stop the server is not always desirable, and servers may support disabling this functionality somehow. The method, and also the exposed keyword, should return True or False depending on whether stopping is allowed or not. That makes it possible for external tools to know if stopping the server succeeded.

[**Using get\_keyword\_names and keyword specific getters**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-580)

This section explains how the Remote library gets keyword names and other information when the server implements get\_keyword\_names. The next sections covers using the newer get\_library\_info method.

The get\_keyword\_names method must return names of the keyword the server contains as a list of strings. Remote servers can, and should, also implement get\_keyword\_arguments, get\_keyword\_types, get\_keyword\_tags and get\_keyword\_documentation methods to provide more information about the keywords. All these methods take the name of the keyword as an argument and what they must return is explained in the table below.

| *Keyword specific getter methods* | |
| --- | --- |
| **Method** | **Return value** |
| get\_keyword\_arguments | Arguments as a list of strings in the [same format as with dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments). |
| get\_keyword\_types | Type information as a list or dictionary of strings. See below for details. |
| get\_keyword\_documentation | Documentation as a string. |
| get\_keyword\_tags | Tags as a list of strings. |

Type information used for [argument conversion](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-conversion) can be returned either as a list mapping type names to arguments based on position or as a dictionary mapping argument names to type names directly. In practice this works the same way as when [specifying types using the @keyword decorator](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-keyword-decorator) with normal libraries. The difference is that because the XML-RPC protocol does not support arbitrary values, type information needs to be specified using type names or aliases like 'int' or 'integer', not using actual types like int. Additionally None or null values may not be allowed by the XML-RPC server, but an empty string can be used to indicate that certain argument does not have type information instead.

Argument conversion is supported also based on default values using the [same logic as with normal libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values). For this to work, arguments with default values must be returned as tuples, not as strings, the [same way as with dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments). For example, argument conversion works if argument information is returned like [('count', 1), ('caseless', True)] but not if it is ['count=1', 'caseless=True'].

Remote servers can also provide [general library documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-general-library-documentation) to be used when generating documentation with the [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc) tool. This information is got by calling get\_keyword\_documentation with special values \_\_intro\_\_ and \_\_init\_\_.

**Note**

get\_keyword\_types is new in Robot Framework 3.1 and support for argument conversion based on defaults is new in Robot Framework 4.0.

[**Using get\_library\_information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-581)

The get\_library\_information method allows returning information about the whole library in one XML-RPC call. The information must be returned as a dictionary where keys are keyword names and values are nested dictionaries containing keyword information. The dictionary can also contain separate entries for generic library information.

The keyword information dictionary can contain keyword arguments, documentation, tags and types, and the respective keys are args, doc, tags and types. Information must be provided using same semantics as when get\_keyword\_arguments, get\_keyword\_documentation, get\_keyword\_tags and get\_keyword\_types discussed in the previous section. If some information is not available, it can be omitted from the info dictionary altogether.

get\_library\_information supports also returning general library documentation to be used with [Libdoc](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "libdoc). It is done by including special \_\_intro\_\_ and \_\_init\_\_ entries into the returned library information dictionary.

For example, a Python library like

*"""Library documentation."""*

**from** **robot.api.deco** **import** keyword

@keyword(tags=['x', 'y'])

**def** example(a: int, b=**True**):

*"""Keyword documentation."""*

**pass**

**def** another():

**pass**

could be mapped into this kind of library information dictionary:

{

'\_\_intro\_\_': {'doc': 'Library documentation'}

'example': {'args': ['a', 'b=True'],

'types': ['int'],

'doc': 'Keyword documentation.',

'tags': ['x', 'y']}

'another: {'args': []}

}

**Note**

get\_library\_information is new in Robot Framework 4.0.

[**Executing remote keywords**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-582)

When the Remote library wants the server to execute some keyword, it calls the remote server's run\_keyword method and passes it the keyword name, a list of arguments, and possibly a dictionary of [free named arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#different-argument-syntaxes). Base types can be used as arguments directly, but more complex types are [converted to supported types](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-argument-and-return-value-types).

The server must return results of the execution in a result dictionary (or map, depending on terminology) containing items explained in the following table. Notice that only the status entry is mandatory, others can be omitted if they are not applicable.

| *Entries in the remote result dictionary* | |
| --- | --- |
| **Name** | **Explanation** |
| status | Mandatory execution status. Either PASS or FAIL. |
| output | Possible output to write into the log file. Must be given as a single string but can contain multiple messages and different [log levels](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#logging-information) in format \*INFO\* First message\n\*HTML\* <b>2nd</b>\n\*WARN\* Another message. It is also possible to embed [timestamps](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamps) to the log messages like \*INFO:1308435758660\* Message with timestamp. |
| return | Possible return value. Must be one of the [supported types](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-argument-and-return-value-types). |
| error | Possible error message. Used only when the execution fails. |
| traceback | Possible stack trace to [write into the log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reporting-keyword-status) using DEBUG level when the execution fails. |
| continuable | When set to True, or any value considered True in Python, the occurred failure is considered [continuable](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#continue-on-failure). |
| fatal | Like continuable, but denotes that the occurred failure is [fatal](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-test-execution-gracefully). |

[**Different argument syntaxes**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-583)

The Remote library is a [dynamic library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library), and in general it handles different argument syntaxes [according to the same rules](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-keyword-arguments) as any other dynamic library. This includes mandatory arguments, default values, varargs, as well as [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax-with-dynamic-libraries).

Also free named arguments (\*\*kwargs) works mostly the [same way as with other dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-named-arguments-with-dynamic-libraries). First of all, the get\_keyword\_arguments must return an argument specification that contains \*\*kwargs exactly like with any other dynamic library. The main difference is that remote servers' run\_keyword method must have an **optional** third argument that gets the kwargs specified by the user. The third argument must be optional because, for backwards-compatibility reasons, the Remote library passes kwargs to the run\_keyword method only when they have been used in the test data.

In practice run\_keyword should look something like the following Python and Java examples, depending on how the language handles optional arguments.

**def** run\_keyword(name, args, kwargs=**None**):

*# ...*

**public** Map run\_keyword(String name, List args) {

*// ...*

}

**public** Map run\_keyword(String name, List args, Map kwargs) {

*// ...*

}

[**4.3   Listener interface**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-502)

Robot Framework has a listener interface that can be used to receive notifications about test execution. Example usages include external test monitors, sending a mail message when a test fails, and communicating with other systems. Listener API version 3 also makes it possible to modify tests and results during the test execution.

Listeners are classes or modules with certain special methods. Listeners that monitor the whole test execution must be taken into use from the command line. In addition to that, [test libraries can register listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libraries-as-listeners) that receive notifications while that library is active.

* [4.3.1   Taking listeners into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-listeners-into-use)
* [4.3.2   Listener interface versions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface-versions)
* [4.3.3   Listener interface methods](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface-methods)
  + [Listener version 2](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-2)
  + [Listener version 3](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-3)
* [4.3.4   Listeners logging](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listeners-logging)
* [4.3.5   Listener examples](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-examples)
  + [Getting information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-information)
  + [Modifying execution and results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#modifying-execution-and-results)
* [4.3.6   Libraries as listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libraries-as-listeners)
  + [Registering listener](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#registering-listener)
  + [Called listener methods](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#called-listener-methods)

[**4.3.1   Taking listeners into use**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-584)

Listeners are taken into use from the command line with the --listener option so that the name of the listener is given to it as an argument. The listener name is got from the name of the class or module implementing the listener, similarly as [library name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-name) is got from the class or module implementing the library. The specified listeners must be in the same [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path) where test libraries are searched from when they are imported. Other option is to give an absolute or a relative path to the listener file [similarly as with test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-physical-path-to-library). It is possible to take multiple listeners into use by using this option several times:

robot --listener MyListener tests.robot

robot --listener com.company.package.Listener tests.robot

robot --listener path/to/MyListener.py tests.robot

robot --listener module.Listener --listener AnotherListener tests.robot

It is also possible to give arguments to listener classes from the command line. Arguments are specified after the listener name (or path) using a colon (:) as a separator. If a listener is given as an absolute Windows path, the colon after the drive letter is not considered a separator. Additionally it is possible to use a semicolon (;) as an alternative argument separator. This is useful if listener arguments themselves contain colons, but requires surrounding the whole value with quotes on UNIX-like operating systems:

robot --listener listener.py:arg1:arg2 tests.robot

robot --listener "listener.py;arg:with:colons" tests.robot

robot --listener c:\path\listener.py;d:\first\arg;e:\second\arg tests.robot

In addition to passing arguments one-by-one as positional arguments, it is possible to pass them using the [named argument syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#named-argument-syntax) exactly when using keywords:

robot --listener listener.py:name=value tests.robot

robot --listener "listener.py;name=value:with:colons;another=value" tests.robot

Listener arguments are automatically converted using [same rules as with keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions) based on [type hints](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#specifying-argument-types-using-function-annotations) and [default values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#implicit-argument-types-based-on-default-values). For example, this listener

**class** **Listener**:

**def** \_\_init\_\_(self, port: int, log=**True**):

self.port = post

self.log = log

could be used like

robot --listener Listener:8270:false

and the first argument would be converted to an integer based on the type hint and the second to a Boolean based on the default value.

**Note**

Both named argument syntax and argument conversion are new in Robot Framework 4.0.

[**4.3.2   Listener interface versions**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-585)

There are two supported listener interface versions. A listener must have an attribute ROBOT\_LISTENER\_API\_VERSION with value 2 or 3, either as a string or as an integer, depending on which API version it uses.

The main difference between listener versions 2 and 3 is that the former only gets information about the execution but cannot directly affect it. The latter interface gets data and result objects Robot Framework itself uses and is thus able to alter execution and change results. See [listener examples](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-examples) for more information about what listeners can do.

[**4.3.3   Listener interface methods**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-586)

Robot Framework creates instances of listener classes when the test execution starts and uses listeners implemented as modules directly. During the test execution different listener methods are called when test suites, test cases and keywords start and end. Additional methods are called when a library or a resource or variable file is imported, when output files are ready, and finally when the whole test execution ends. A listener is not required to implement any official interface, and it only needs to have the methods it actually needs.

Listener versions 2 and 3 have mostly the same methods, but the arguments they accept are different. These methods and their arguments are explained in the following sections. All methods that have an underscore in their name have also *camelCase* alternative. For example, start\_suite method can be used also with name startSuite.

[**Listener version 2**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-587)

Listener methods in the API version 2 are listed in the following table. All methods related to test execution progress have the same signature method(name, attributes), where attributes is a dictionary containing details of the event. Listener methods are free to do whatever they want to do with the information they receive, but they cannot directly change it. If that is needed, [listener version 3](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-3) can be used instead.

| *Methods in the listener API 2* | | |
| --- | --- | --- |
| **Method** | **Arguments** | **Documentation** |
| start\_suite | name, attributes | Called when a test suite starts.  Contents of the attribute dictionary:   * id: Suite id. s1 for the top level suite, s1-s1 for its first child suite, s1-s2 for the second child, and so on. * longname: Suite name including parent suites. * doc: Suite documentation. * metadata: [Free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) as a dictionary/map. * source: An absolute path of the file/directory the suite was created from. * suites: Names of the direct child suites this suite has as a list. * tests: Names of the tests this suite has as a list. Does not include tests of the possible child suites. * totaltests: The total number of tests in this suite. and all its sub-suites as an integer. * starttime: Suite execution start time. |
| end\_suite | name, attributes | Called when a test suite ends.  Contents of the attribute dictionary:   * id: Same as in start\_suite. * longname: Same as in start\_suite. * doc: Same as in start\_suite. * metadata: Same as in start\_suite. * source: Same as in start\_suite. * starttime: Same as in start\_suite. * endtime: Suite execution end time. * elapsedtime: Total execution time in milliseconds as an integer * status: Suite status as string PASS, FAIL or SKIP. * statistics: Suite statistics (number of passed and failed tests in the suite) as a string. * message: Error message if suite setup or teardown has failed, empty otherwise. |
| start\_test | name, attributes | Called when a test case starts.  Contents of the attribute dictionary:   * id: Test id in format like s1-s2-t2, where the beginning is the parent suite id and the last part shows test index in that suite. * longname: Test name including parent suites. * originalname: Test name with possible variables unresolved. New in RF 3.2. * doc: Test documentation. * tags: Test tags as a list of strings. * template: The name of the template used for the test. An empty string if the test not templated. * source: An absolute path of the test case source file. New in RF 4.0. * lineno: Line number where the test starts in the source file. New in RF 3.2. * starttime: Test execution execution start time. |
| end\_test | name, attributes | Called when a test case ends.  Contents of the attribute dictionary:   * id: Same as in start\_test. * longname: Same as in start\_test. * originalname: Same as in start\_test. * doc: Same as in start\_test. * tags: Same as in start\_test. * template: Same as in start\_test. * source: Same as in start\_test. * lineno: Same as in start\_test. * starttime: Same as in start\_test. * endtime: Test execution execution end time. * elapsedtime: Total execution time in milliseconds as an integer * status: Test status as string PASS, FAIL or SKIP. * message: Status message. Normally an error message or an empty string. |
| start\_keyword | name, attributes | Called when a keyword or a control structure such as IF/ELSE or TRY/EXCEPT starts.  With keywords name is the full keyword name containing possible library or resource name as a prefix like MyLibrary.Example Keyword. With control structures name contains string representation of parameters.  Contents of the attribute dictionary:   * type: String specifying type of the started item. Possible values are: KEYWORD, SETUP, TEARDOWN, FOR, WHILE, ITERATION, IF, ELSE IF, ELSE, TRY, EXCEPT, FINALLY, RETURN, BREAK and CONTINUE. All type values were changed in RF 4.0 and in RF 5.0 FOR ITERATION was changed to ITERATION. * kwname: Name of the keyword without library or resource prefix. String representation of parameters with control structures. * libname: Name of the library or resource file the keyword belongs to. An empty string with user keywords in a test case file and with control structures. * doc: Keyword documentation. * args: Keyword's arguments as a list of strings. * assign: A list of variable names that keyword's return value is assigned to. * tags: [Keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags) as a list of strings. * source: An absolute path of the file where the keyword was used. New in RF 4.0. * lineno: Line where the keyword was used. New in RF 4.0. * status: Initial keyword status. NOT RUN if keyword is not executed (e.g. due to an earlier failure), NOT SET otherwise. New in RF 4.0. * starttime: Keyword execution start time. |
| end\_keyword | name, attributes | Called when a keyword ends.  name is the full keyword name containing possible library or resource name as a prefix. For example, MyLibrary.Example Keyword.  Contents of the attribute dictionary:   * type: Same as with start\_keyword. * kwname: Same as with start\_keyword. * libname: Same as with start\_keyword. * doc: Same as with start\_keyword. * args: Same as with start\_keyword. * assign: Same as with start\_keyword. * tags: Same as with start\_keyword. * source: Same as with start\_keyword. * lineno: Same as with start\_keyword. * starttime: Same as with start\_keyword. * endtime: Keyword execution end time. * elapsedtime: Total execution time in milliseconds as an integer * status: Keyword status as string PASS, FAIL, SKIP or NOT RUN. SKIP and NOT RUN are new in RF 4.0. |
| log\_message | message | Called when an executed keyword writes a log message.  message is a dictionary with the following contents:   * message: The content of the message. * level: [Log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels) used in logging the message. * timestamp: Message creation time in format YYYY-MM-DD hh:mm:ss.mil. * html: String yes or no denoting whether the message should be interpreted as HTML or not.   Not called if the message level is below the current [threshold level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). |
| message | message | Called when the framework itself writes a [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) message.  message is a dictionary with the same contents as with log\_message method. |
| library\_import | name, attributes | Called when a library has been imported.  name is the name of the imported library. If the library has been imported using the [WITH NAME syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-custom-name-to-test-library), name is the specified alias.  Contents of the attribute dictionary:   * args: Arguments passed to the library as a list. * originalname: The original library name when using the WITH NAME syntax, otherwise same as name. * source: An absolute path to the library source. None if getting the source of the library failed for some reason. * importer: An absolute path to the file importing the library. None when [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) is imported well as when using the *Import Library* keyword. |
| resource\_import | name, attributes | Called when a resource file has been imported.  name is the name of the imported resource file without the file extension.  Contents of the attribute dictionary:   * source: An absolute path to the imported resource file. * importer: An absolute path to the file importing the resource file. None when using the *Import Resource* keyword. |
| variables\_import | name, attributes | Called when a variable file has been imported.  name is the name of the imported variable file with the file extension.  Contents of the attribute dictionary:   * args: Arguments passed to the variable file as a list. * source: An absolute path to the imported variable file. * importer: An absolute path to the file importing the resource file. None when using the *Import Variables* keyword. |
| output\_file | path | Called when writing to an [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) is ready.  path is an absolute path to the file. |
| log\_file | path | Called when writing to a [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) is ready.  path is an absolute path to the file. |
| report\_file | path | Called when writing to a [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) is ready.  path is an absolute path to the file. |
| xunit\_file | path | Called when writing to an [xunit file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit-file) is ready.  path is an absolute path to the file. |
| debug\_file | path | Called when writing to a [debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file) is ready.  path is an absolute path to the file. |
| close |  | Called when the whole test execution ends.  With [library listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-listeners) called when the library goes out of scope. |

[**Listener version 3**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-588)

Listener version 3 has mostly the same methods as [listener version 2](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-2) but arguments of the methods related to test execution are different. This API gets actual running and result model objects used by Robot Framework itself, and listeners can both directly query information they need and also change the model objects on the fly.

Listener version 3 does not yet have all methods that the version 2 has. The main reason is that [suitable model objects are not available internally](https://github.com/robotframework/robotframework/issues/1208#issuecomment-164910769). The close method and methods related to output files are called exactly same way in both versions.

| *Methods in the listener API 3* | | |
| --- | --- | --- |
| **Method** | **Arguments** | **Documentation** |
| start\_suite | data, result | Called when a test suite starts.  data and result are model objects representing the [executed test suite](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite) and [its execution results](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.TestSuite), respectively. |
| end\_suite | data, result | Called when a test suite ends.  Same arguments as with start\_suite. |
| start\_test | data, result | Called when a test case starts.  data and result are model objects representing the [executed test case](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestCase) and [its execution results](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.TestCase), respectively. |
| end\_test | data, result | Called when a test case ends.  Same arguments as with start\_test. |
| start\_keyword | N/A | Not currently implemented. |
| end\_keyword | N/A | Not currently implemented. |
| log\_message | message | Called when an executed keyword writes a log message. message is a model object representing the [logged message](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.Message).  This method is not called if the message has level below the current [threshold level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels). |
| message | message | Called when the framework itself writes a [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) message.  message is same object as with log\_message. |
| library\_import | N/A | Not currently implemented. |
| resource\_import | N/A | Not currently implemented. |
| variables\_import | N/A | Not currently implemented. |
| output\_file | path | Called when writing to an [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) is ready.  path is an absolute path to the file. |
| log\_file | path | Called when writing to a [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) is ready.  path is an absolute path to the file. |
| report\_file | path | Called when writing to a [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file) is ready.  path is an absolute path to the file. |
| xunit\_file | path | Called when writing to an [xunit file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit-file) is ready.  path is an absolute path to the file. |
| debug\_file | path | Called when writing to a [debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file) is ready.  path is an absolute path to the file. |
| close |  | Called when the whole test execution ends.  With [library listeners](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-listeners) called when the library goes out of scope. |

[**4.3.4   Listeners logging**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-589)

Robot Framework offers a [programmatic logging APIs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-logging-apis) that listeners can utilize. There are some limitations, however, and how different listener methods can log messages is explained in the table below.

| *How listener methods can log* | |
| --- | --- |
| **Methods** | **Explanation** |
| start\_keyword, end\_keyword, log\_message | Messages are logged to the normal [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file) under the executed keyword. |
| start\_suite, end\_suite, start\_test, end\_test | Messages are logged to the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log). Warnings are shown also in the [execution errors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#errors-and-warnings-during-execution) section of the normal log file. |
| message | Messages are normally logged to the syslog. If this method is used while a keyword is executing, messages are logged to the normal log file. |
| Other methods | Messages are only logged to the syslog. |

**Note**

To avoid recursion, messages logged by listeners are not sent to listener methods log\_message and message.

[**4.3.5   Listener examples**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-590)

This section contains examples using the listener interface. There are first examples that just receive information from Robot Framework and then examples that modify executed tests and created results.

[**Getting information**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-591)

The first example is implemented as Python module and uses the [listener version 2](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-2).

*"""Listener that stops execution if a test fails."""*

ROBOT\_LISTENER\_API\_VERSION = 2

**def** end\_test(name, attrs):

**if** attrs['status'] == 'FAIL':

print('Test "**%s**" failed: **%s**' % (name, attrs['message']))

input('Press enter to continue.')

If the above example would be saved to, for example, *PauseExecution.py* file, it could be used from the command line like this:

robot --listener path/to/PauseExecution.py tests.robot

The same example could also be implemented also using the newer [listener version 3](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-3) and used exactly the same way from the command line.

*"""Listener that stops execution if a test fails."""*

ROBOT\_LISTENER\_API\_VERSION = 3

**def** end\_test(data, result):

**if** **not** result.passed:

print('Test "**%s**" failed: **%s**' % (result.name, result.message))

input('Press enter to continue.')

The next example, which still uses Python, is slightly more complicated. It writes all the information it gets into a text file in a temporary directory without much formatting. The filename may be given from the command line, but also has a default value. Note that in real usage, the [debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file) functionality available through the command line option --debugfile is probably more useful than this example.

**import** **os.path**

**import** **tempfile**

**class** **PythonListener**:

ROBOT\_LISTENER\_API\_VERSION = 2

**def** \_\_init\_\_(self, filename='listen.txt'):

outpath = os.path.join(tempfile.gettempdir(), filename)

self.outfile = open(outpath, 'w')

**def** start\_suite(self, name, attrs):

self.outfile.write("**%s** '**%s**'**\n**" % (name, attrs['doc']))

**def** start\_test(self, name, attrs):

tags = ' '.join(attrs['tags'])

self.outfile.write("- **%s** '**%s**' [ **%s** ] :: " % (name, attrs['doc'], tags))

**def** end\_test(self, name, attrs):

**if** attrs['status'] == 'PASS':

self.outfile.write('PASS**\n**')

**else**:

self.outfile.write('FAIL: **%s\n**' % attrs['message'])

**def** end\_suite(self, name, attrs):

self.outfile.write('**%s\n%s\n**' % (attrs['status'], attrs['message']))

**def** close(self):

self.outfile.close()

[**Modifying execution and results**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-592)

These examples illustrate how to modify the executed tests and suites as well as the execution results. All these examples require using the [listener version 3](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-version-3).

**Modifying executed suites and tests**

Changing what is executed requires modifying the model object containing the executed [test suite](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestSuite) or [test case](http://robot-framework.readthedocs.org/en/master/autodoc/robot.running.html#robot.running.model.TestCase) objects passed as the first argument to start\_suite and start\_test methods. This is illustrated by the example below that adds a new test to each executed test suite and a new keyword to each test.

ROBOT\_LISTENER\_API\_VERSION = 3

**def** start\_suite(suite, result):

suite.tests.create(name='New test')

**def** start\_test(test, result):

test.keywords.create(name='Log', args=['Keyword added by listener!'])

Trying to modify execution in end\_suite or end\_test methods does not work, simply because that suite or test has already been executed. Trying to modify the name, documentation or other similar metadata of the current suite or test in start\_suite or start\_test method does not work either, because the corresponding result object has already been created. Only changes to child tests or keywords actually have an effect.

This API is very similar to the [pre-run modifier](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pre-run-modifier) API that can be used to modify suites and tests before the whole test execution starts. The main benefit of using the listener API is that modifications can be done dynamically based on execution results or otherwise. This allows, for example, interesting possibilities for model based testing.

Although the listener interface is not built on top of Robot Framework's internal [visitor interface](http://robot-framework.readthedocs.org/en/master/autodoc/robot.model.html#module-robot.model.visitor) similarly as the pre-run modifier API, listeners can still use the visitors interface themselves. For example, the SelectEveryXthTest visitor used in [pre-run modifier](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pre-run-modifier) examples could be used like this:

**from** **SelectEveryXthTest** **import** SelectEveryXthTest

ROBOT\_LISTENER\_API\_VERSION = 3

**def** start\_suite(suite, result):

selector = SelectEveryXthTest(x=2)

suite.visit(selector)

**Modifying results**

Test execution results can be altered by modifying [test suite](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.TestSuite) and [test case](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.TestCase) result objects passed as the second argument to start\_suite and start\_test methods, respectively, and by modifying the [message](http://robot-framework.readthedocs.org/en/master/autodoc/robot.result.html#robot.result.model.Message) object passed to the log\_message method. This is demonstrated by the following listener that is implemented as a class.

**class** **ResultModifier**:

ROBOT\_LISTENER\_API\_VERSION = 3

**def** \_\_init\_\_(self, max\_seconds=10):

self.max\_milliseconds = float(max\_seconds) \* 1000

**def** start\_suite(self, data, suite):

suite.doc = 'Documentation set by listener.'

*# Information about tests only available via data at this point.*

smoke\_tests = [test **for** test **in** data.tests **if** 'smoke' **in** test.tags]

suite.metadata['Smoke tests'] = len(smoke\_tests)

**def** end\_test(self, data, test):

**if** test.status == 'PASS' **and** test.elapsedtime > self.max\_milliseconds:

test.status = 'FAIL'

test.message = 'Test execution took too long.'

**def** log\_message(self, msg):

**if** msg.level == 'WARN' **and** **not** msg.html:

msg.message = '<b style="font-size: 1.5em">**%s**</b>' % msg.message

msg.html = **True**

A limitation is that modifying the name of the current test suite or test case is not possible because it has already been written to the [output.xml](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-xml) file when listeners are called. Due to the same reason modifying already finished tests in the end\_suite method has no effect either.

This API is very similar to the [pre-Rebot modifier](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pre-rebot-modifier) API that can be used to modify results before report and log are generated. The main difference is that listeners modify also the created *output.xml* file.

[**4.3.6   Libraries as listeners**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-593)

Sometimes it is useful also for [test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries) to get notifications about test execution. This allows them, for example, to perform certain clean-up activities automatically when a test suite or the whole test execution ends.

[**Registering listener**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-594)

A test library can register a listener by using ROBOT\_LIBRARY\_LISTENER attribute. The value of this attribute should be an instance of the listener to use. It may be a totally independent listener or the library itself can act as a listener. To avoid listener methods to be exposed as keywords in the latter case, it is possible to prefix them with an underscore. For example, instead of using end\_suite it is possible to use \_end\_suite.

Following examples illustrates using an external listener as well as library acting as a listener itself:

**from** **listener** **import** Listener

**class** **LibraryWithExternalListener**:

ROBOT\_LIBRARY\_LISTENER = Listener()

ROBOT\_LIBRARY\_SCOPE = 'GLOBAL'

ROBOT\_LISTENER\_API\_VERSION = 2

// actual library code here ...

}

**class** **PythonLibraryAsListenerItself**:

ROBOT\_LIBRARY\_SCOPE = 'TEST SUITE'

ROBOT\_LISTENER\_API\_VERSION = 2

**def** \_\_init\_\_(self):

self.ROBOT\_LIBRARY\_LISTENER = self

**def** \_end\_suite(self, name, attrs):

print('Suite **%s** (**%s**) ending.' % (name, attrs['id']))

*# actual library code here ...*

As the seconds example above already demonstrated, library listeners have to specify [listener interface versions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface-versions) using ROBOT\_LISTENER\_API\_VERSION attribute exactly like any other listener.

It is also possible to specify multiple listeners for a single library by giving ROBOT\_LIBRARY\_LISTENER a value as a list:

**from** **listeners** **import** Listener1, Listener2, Listener3

**class** **LibraryWithMultipleListeners**:

ROBOT\_LIBRARY\_LISTENER = [Listener1(), Listener2(), Listener3()]

*# actual library code here ...*

[**Called listener methods**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-595)

Library's listener will get notifications about all events in suites where the library is imported. In practice this means that start\_suite, end\_suite, start\_test, end\_test, start\_keyword, end\_keyword, log\_message, and message methods are called inside those suites.

If the library creates a new listener instance every time when the library itself is instantiated, the actual listener instance to use will change according to the [library scope](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-scope). In addition to the previously listed listener methods, close method is called when the library goes out of the scope.

See [Listener interface methods](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface-methods) section above for more information about all these methods.

[**5   Supporting Tools**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-141)

* [5.1   Library documentation tool (Libdoc)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-documentation-tool-libdoc)
* [5.2   Test data documentation tool (Testdoc)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-documentation-tool-testdoc)
* [5.3   Test data clean-up tool (Tidy)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-clean-up-tool-tidy)
* [5.4   External tools](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#external-tools)

[**5.1   Library documentation tool (Libdoc)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-596)

* [5.1.1   General usage](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#general-usage)
* [5.1.2   Writing documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#writing-documentation)
* [5.1.3   Documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-syntax)
* [5.1.4   Internal linking](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#internal-linking-1)
* [5.1.5   Representing arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#representing-arguments)
* [5.1.6   Libdoc example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libdoc-example)

Libdoc is Robot Framework's built-in tool that can generate documentation for Robot Framework libraries and resource files. It can generate HTML documentation for humans as well as machine readable spec files in XML and JSON formats. Libdoc also has few special commands to show library or resource information on the console.

Documentation can be created for:

* libraries implemented using the normal static library [API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#python-libraries),
* libraries using the [dynamic API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-libraries), including remote libraries, and
* [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files).

Additionally it is possible to use XML and JSON spec files created by Libdoc earlier as an input.

**Note**

The support for the JSON spec files is new in Robot Framework 4.0.

[**5.1.1   General usage**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-600)

**Synopsis**

libdoc [options] library\_or\_resource output\_file

libdoc [options] library\_or\_resource list|show|version [names]

**Options**

|  |  |
| --- | --- |
| -f, --format *<html|xml|json|libspec>* | |
|  | Specifies whether to generate an HTML output for humans or a machine readable spec file in XML or JSON format. The libspec format means XML spec with documentations converted to HTML. The default format is got from the output file extension. |
| -s, --specdocformat *<raw|html>* | |
|  | Specifies the documentation format used with XML and JSON spec files. raw means preserving the original documentation format and html means converting documentation to HTML. The default is raw with XML spec files and html with JSON specs and when using the special libspec format. New in Robot Framework 4.0. |
| -F, --docformat *<robot|html|text|rest>* | |
|  | Specifies the source documentation format. Possible values are Robot Framework's documentation format, HTML, plain text, and reStructuredText. Default value can be specified in test library source code and the initial default value is robot. |
| -N, --name *<newname>* | |
|  | Sets the name of the documented library or resource. |
| -V, --version *<newversion>* | |
|  | Sets the version of the documented library or resource. The default value for test libraries is [defined in the source code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#library-version). |
| -P, --pythonpath *<path>* | |
|  | Additional locations where to search for libraries and resources similarly as when [running tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-pythonpath-option). |
| --quiet | Do not print the path of the generated output file to the console. New in Robot Framework 4.0. |
| -h, --help | Prints this help. |

**Executing Libdoc**

The easiest way to run Libdoc is using the libdoc command created as part of the normal installation:

libdoc ExampleLibrary ExampleLibrary.html

Alternatively it is possible to execute the robot.libdoc module directly. This approach is especially useful if you have installed Robot Framework using multiple Python versions and want to use a specific version with Libdoc:

python -m robot.libdoc ExampleLibrary ExampleLibrary.html

python3.9 -m robot.libdoc ExampleLibrary ExampleLibrary.html

Yet another alternative is running the robot.libdoc module as a script:

python path/to/robot/libdoc.py ExampleLibrary ExampleLibrary.html

**Note**

The separate libdoc command is new in Robot Framework 4.0.

**Specifying library or resource file**

**Python libraries and dynamic libraries with name or path**

When documenting libraries implemented with Python or that use the [dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api), it is possible to specify the library either by using just the library name or path to the library source code:

libdoc ExampleLibrary ExampleLibrary.html

libdoc src/ExampleLibrary.py docs/ExampleLibrary.html

In the former case the library is searched using the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path) and its name must be in the same format as when [importing libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-libraries) in Robot Framework test data.

If these libraries require arguments when they are imported, the arguments must be catenated with the library name or path using two colons like MyLibrary::arg1::arg2. If arguments change what keywords the library provides or otherwise alter its documentation, it might be a good idea to use --name option to also change the library name accordingly.

**Resource files with path**

Resource files must always be specified using a path:

libdoc example.resource example.html

If the path does not exist, resource files are also searched from all directories in the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path) similarly as when executing test cases.

**Libdoc spec files**

Earlier generated Libdoc XML or JSON spec files can also be used as inputs. This works if spec files use either *\*.xml*, *\*.libspec* or *\*.json* extension:

libdoc Example.xml Example.html

libdoc Example.libspec Example.html

libdoc Example.json Example.html

**Note**

Support for the *\*.libspec* extension is new in Robot Framework 3.2.

**Note**

Support for the *\*.json* extension is new in Robot Framework 4.0.

**Generating documentation**

Libdoc can generate documentation in HTML (for humans) and XML or JSON (for tools) formats. The file where to write the documentation is specified as the second argument after the library/resource name or path, and the output format is got from the output file extension by default.

**Libdoc HTML documentation**

Most Robot Framework libraries use Libdoc to generate library documentation in HTML format. This format is thus familiar for most people who have used Robot Framework. A simple example can be seen below, and it has been generated based on the example found a [bit later in this section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#python-libraries).

[텍스트이(가) 표시된 사진

자동 생성된 설명](https://robotframework.org/robotframework/latest/images/ExampleLibrary.html)

The HTML documentation starts with general library introduction, continues with a section about configuring the library when it is imported (when applicable), and finally has shortcuts to all keywords and the keywords themselves. The magnifying glass icon on the lower right corner opens the keyword search dialog that can also be opened by simply pressing the s key.

Libdoc automatically creates HTML documentation if the output file extension is *\*.html*. If there is a need to use some other extension, the format can be specified explicitly with the --format option.

libdoc OperatingSystem OperatingSystem.html

libdoc --name MyLibrary Remote::http://10.0.0.42:8270 MyLibrary.html

libdoc --format HTML test/resource.robot doc/resource.htm

**Libdoc XML spec files**

Libdoc can also generate documentation in XML format that is suitable for external tools such as editors. It contains all the same information as the HTML format but in a machine readable format.

XML spec files also contain library and keyword source information so that the library and each keyword can have source path (source attribute) and line number (lineno attribute). The source path is relative to the directory where the spec file is generated thus does not refer to a correct file if the spec is moved. The source path is omitted with keywords if it is the same as with the library, and both the source path and the line number are omitted if getting them from the library fails for whatever reason.

Libdoc automatically uses the XML format if the output file extension is *\*.xml* or *\*.libspec*. When using the special *\*.libspec* extension, Libdoc automatically enables the options -f XML -s HTML which means creating an XML output file where keyword documentation is converted to HTML. If needed, the format can be explicitly set with the --format option.

libdoc OperatingSystem OperatingSystem.xml

libdoc test/resource.robot doc/resource.libspec

libdoc --format xml MyLibrary MyLibrary.spec

libdoc --format xml -s html MyLibrary MyLibrary.xml

The exact Libdoc spec file format is documented with an [XML schema](https://en.wikipedia.org/wiki/XML_Schema_(W3C)) (XSD) at <https://github.com/robotframework/robotframework/tree/master/doc/schema>. The spec file format may change between Robot Framework major releases.

To make it easier for external tools to know how to parse a certain spec file, the spec file root element has a dedicated specversion attribute. It was added in Robot Framework 3.2 with value 2 and earlier spec files can be considered to have version 1. The spec version will be incremented in the future if and when changes are made. Robot Framework 4.0 introduced new spec version 3 which is incompatible with earlier versions.

**Note**

The XML:HTML format introduced in Robot Framework 3.2. has been replaced by the format LIBSPEC ot the option combination --format XML --specdocformat HTML.

**Note**

Including source information and spec version are new in Robot Framework 3.2.

**Libdoc JSON spec files**

Since Robot Framework 4.0 Libdoc can also generate documentation in JSON format that is suitable for external tools such as editors or web pages. It contains all the same information as the HTML format but in a machine readable format.

Similar to XML spec files the JSON spec files contain all information and can also be used as input to Libdoc. From that format any other output format can be created. By default the library documentation strings are converted to HTML format within the JSON output file.

The exact JSON spec file format is documented with an [JSON schema](https://json-schema.org/) at <https://github.com/robotframework/robotframework/tree/master/doc/schema>. The spec file format may change between Robot Framework major releases.

**Viewing information on console**

Libdoc has three special commands to show information on the console. These commands are used instead of the name of the output file, and they can also take additional arguments.

list

List names of the keywords the library/resource contains. Can be limited to show only certain keywords by passing optional patterns as arguments. Keyword is listed if its name contains given pattern.

show

Show library/resource documentation. Can be limited to show only certain keywords by passing names as arguments. Keyword is shown if its name matches any given name. Special argument intro will show only the library introduction and importing sections.

version

Show library version

Optional patterns given to list and show are case and space insensitive. Both also accept \* and ? as wildcards.

Examples:

libdoc Dialogs list

libdoc SeleniumLibrary list browser

libdoc Remote::10.0.0.42:8270 show

libdoc Dialogs show PauseExecution execute\*

libdoc SeleniumLibrary show intro

libdoc SeleniumLibrary version

[**5.1.2   Writing documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-601)

This section discusses writing documentation for [Python](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#python-libraries) based test libraries that use the static library API as well as for [dynamic libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-libraries) and [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-file-documentation). [Creating test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries) and [resource files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files) is described in more details elsewhere in the User Guide.

**Python libraries**

The documentation for Python libraries that use the [static library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-keywords) is written simply as doc strings for the library class or module and for methods implementing keywords. The first line of the method documentation is considered as a short documentation for the keyword (used, for example, as a tool tip in links in the generated HTML documentation), and it should thus be as describing as possible, but not too long.

The simple example below illustrates how to write the documentation in general. How the HTML documentation generated based on this example looks like can be seen [above](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libdoc-html-documentation), and there is also a [bit longer example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libdoc-example) at the end of this chapter.

**class** **ExampleLibrary**:

*"""Library for demo purposes.*

*This library is only used in an example and it doesn't do anything useful.*

*"""*

**def** my\_keyword(self):

*"""Does nothing."""*

**pass**

**def** your\_keyword(self, arg):

*"""Takes one argument and \*does nothing\* with it.*

*Examples:*

*| Your Keyword | xxx |*

*| Your Keyword | yyy |*

*"""*

**pass**

If you want to use non-ASCII characters in the documentation, the documentation must either be Unicode string (default in Python 3) or UTF-8 encoded bytes.

**Tip**

When using Python 2, you it is a good idea to set the [source code encoding](http://www.python.org/dev/peps/pep-0263) to ease using non-ASCII characters.

For more information on Python documentation strings, see [PEP-257](http://www.python.org/dev/peps/pep-0257).

**Dynamic libraries**

To be able to generate meaningful documentation for dynamic libraries, the libraries must return keyword argument names and documentation using get\_keyword\_arguments and get\_keyword\_documentation methods (or using their camelCase variants getKeywordArguments and getKeywordDocumentation). Libraries can also support general library documentation via special \_\_intro\_\_ and \_\_init\_\_ values to the get\_keyword\_documentation method.

See the [Dynamic library API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dynamic-library-api) section for more information about how to create these methods.

**Importing section**

A separate section about how the library is imported is created based on its initialization methods. If the library has an \_\_init\_\_ method that takes arguments in addition to self, its documentation and arguments are shown.

**class** **TestLibrary**:

**def** \_\_init\_\_(self, mode='default')

*"""Creates new TestLibrary. `mode` argument is used to determine mode."""*

self.mode = mode

**def** some\_keyword(self, arg):

*"""Does something based on given `arg`.*

*What is done depends on the `mode` specified when `importing` the library.*

*"""*

**if** self.mode == 'secret':

*# ...*

**Resource file documentation**

Keywords in resource files can have documentation using *[Documentation]* setting, and this documentation is also used by Libdoc. First line of the documentation (until the first [implicit newline](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#newlines-in-test-data) or explicit \n) is considered to be the short documentation similarly as with test libraries.

Also the resource file itself can have *Documentation* in the Setting section for documenting the whole resource file.

Possible variables in resource files can not be documented.

**\*\*\* Settings \*\*\***

**Documentation** Resource file for demo purposes.

... This resource is only used in an example and it doesn't do anything useful.

**\*\*\* Keywords \*\*\***

**My Keyword**

[**Documentation**] Does nothing

No Operation

**Your Keyword**

[**Arguments**] ${arg}

[**Documentation**] Takes one argument and \*does nothing\* with it.

...

... Examples:

... | Your Keyword | xxx |

... | Your Keyword | yyy |

No Operation

[**5.1.3   Documentation syntax**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-602)

Libdoc supports documentation in Robot Framework's own [documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-syntax-1), HTML, plain text, and [reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText). The format to use can be specified in [library source code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-format) using ROBOT\_LIBRARY\_DOC\_FORMAT attribute or given from the command line using --docformat (-F) option. In both cases the possible case-insensitive values are ROBOT (default), HTML, TEXT and reST.

Robot Framework's own documentation format is the default and generally recommended format. Other formats are especially useful when using existing code with existing documentation in test libraries.

**Robot Framework documentation syntax**

Most important features in Robot Framework's [documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-syntax-1) are formatting using \*bold\* and \_italic\_, custom links and automatic conversion of URLs to links, and the possibility to create tables and pre-formatted text blocks (useful for examples) simply with pipe character. If documentation gets longer, support for section titles can also be handy.

Some of the most important formatting features are illustrated in the example below. Notice that since this is the default format, there is no need to use ROBOT\_LIBRARY\_DOC\_FORMAT attribute nor give the format from the command line.

*"""Example library in Robot Framework format.*

*- Formatting with \*bold\* and \_italic\_.*

*- URLs like http://example.com are turned to links.*

*- Custom links like [http://robotframework.org|Robot Framework] are supported.*

*- Linking to `My Keyword` works.*

*"""*

**def** my\_keyword():

*"""Nothing more to see here."""*

**Creating table of contents automatically**

With bigger libraries it is often useful to add a table of contents to the library introduction. When using the Robot Framework documentation format, this can be done automatically by adding a special %TOC% marker into a line on its own. The table of contents is created based on the top-level [section titles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#section-titles) (e.g. = Section =) used in the introduction. In addition to them, the TOC also gets links to the [automatically created sections](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#linking-to-automatic-sections) for shortcuts and keywords as well as for importing and tags sections when applicable.

*"""Example library demonstrating TOC generation.*

*The %TOC% marker only creates the actual table of contents and possible*

*header or other explanation needs to be added separately like done below.*

*== Table of contents ==*

*%TOC%*

*= Section title =*

*The top-level section titles are automatically added to the TOC.*

*= Second section =*

*== Sub section ==*

*Sub section titles are not added to the TOC.*

*"""*

**def** my\_keyword():

*"""Nothing more to see here."""*

**Note**

Automatic TOC generation is a new feature in Robot Framework 3.2.

**HTML documentation syntax**

When using HTML format, you can create documentation pretty much freely using any syntax. The main drawback is that HTML markup is not that human friendly, and that can make the documentation in the source code hard to maintain and read. Documentation in HTML format is used by Libdoc directly without any transformation or escaping. The special syntax for [linking to keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#linking-to-keywords) using syntax like `My Keyword` is supported, however.

Example below contains the same formatting examples as the previous example. Now ROBOT\_LIBRARY\_DOC\_FORMAT attribute must be used or format given on the command line like --docformat HTML.

*"""Example library in HTML format.*

*<ul>*

*<li>Formatting with <b>bold</b> and <i>italic</i>.*

*<li>URLs are not turned to links automatically.*

*<li>Custom links like <a href="http://www.w3.org/html">HTML</a> are supported.*

*<li>Linking to `My Keyword` works.*

*</ul>*

*"""*

ROBOT\_LIBRARY\_DOC\_FORMAT = 'HTML'

**def** my\_keyword():

*"""Nothing more to see here."""*

**Plain text documentation syntax**

When the plain text format is used, Libdoc uses the documentation as-is. Newlines and other whitespace are preserved except for indentation, and HTML special characters (<>&) escaped. The only formatting done is turning URLs into clickable links and supporting [internal linking](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#internal-linking) like `My Keyword`.

*"""Example library in plain text format.*

*- Formatting is not supported.*

*- URLs like http://example.com are turned to links.*

*- Custom links are not supported.*

*- Linking to `My Keyword` works.*

*"""*

ROBOT\_LIBRARY\_DOC\_FORMAT = 'text'

**def** my\_keyword():

*"""Nothing more to see here."""*

**reStructuredText documentation syntax**

[reStructuredText](https://en.wikipedia.org/wiki/ReStructuredText) is simple yet powerful markup syntax used widely in Python projects (including this User Guide) and elsewhere. The main limitation is that you need to have the [docutils](https://pypi.python.org/pypi/docutils) module installed to be able to generate documentation using it. Because backtick characters have special meaning in reStructuredText, [linking to keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#linking-to-keywords) requires them to be escaped like \`My Keyword\`.

One of the nice features that reStructured supports is the ability to mark code blocks that can be syntax highlighted. Syntax highlight requires additional [Pygments](http://pygments.org/) module and supports all the languages that Pygments supports.

*"""Example library in reStructuredText format.*

*- Formatting with \*\*bold\*\* and \*italic\*.*

*- URLs like http://example.com are turned to links.*

*- Custom links like reStructuredText\_\_ are supported.*

*- Linking to \`My Keyword\` works but requires backtics to be escaped.*

*\_\_ http://docutils.sourceforge.net*

*.. code:: robotframework*

*\*\*\* Test Cases \*\*\**

*Example*

*My keyword # How cool is this!!?!!?!1!!*

*"""*

ROBOT\_LIBRARY\_DOC\_FORMAT = 'reST'

**def** my\_keyword():

*"""Nothing more to see here."""*

[**5.1.4   Internal linking**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-603)

Libdoc supports internal linking to keywords and different sections in the documentation. Linking is done by surrounding the target name with backtick characters like `target`. Target names are case-insensitive and possible targets are explained in the subsequent sections.

There is no error or warning if a link target is not found, but instead Libdoc just formats the text in italics. Earlier this formatting was recommended to be used when referring to keyword arguments, but that was problematic because it could accidentally create internal links. Nowadays it is recommended to use [inline code style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-styles) with double backticks like ``argument`` instead. The old formatting of single backticks may even be removed in the future in favor of giving an error when a link target is not found.

In addition to the examples in the following sections, internal linking and argument formatting is shown also in the [longer example](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libdoc-example) at the end of this chapter.

**Linking to keywords**

All keywords the library have automatically create link targets and they can be linked using syntax `Keyword Name`. This is illustrated with the example below where both keywords have links to each others.

**def** keyword(log\_level="INFO"):

*"""Does something and logs the output using the given level.*

*Valid values for log level` are "INFO" (default) "DEBUG" and "TRACE".*

*See also `Another Keyword`.*

*"""*

*# ...*

**def** another\_keyword(argument, log\_level="INFO"):

*"""Does something with the given argument else and logs the output.*

*See `Keyword` for information about valid log levels.*

*"""*

*# ...*

**Note**

When using [reStructuredText documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "restructuredtext-documentation-syntax), backticks must be escaped like \`Keyword Name\`.

**Linking to automatic sections**

The documentation generated by Libdoc always contains sections for overall library introduction and for keywords. If a library itself takes arguments, there is also separate [importing section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-section). If any of the keywords has [tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-tags), a separate selector for them is also shown in the overview.

All the sections act as targets that can be linked, and the possible target names are listed in the table below. Using these targets is shown in the example of the next section.

| *Automatic section link targets* | |
| --- | --- |
| **Section** | **Target** |
| Introduction | `introduction` and `library introduction` |
| Importing | `importing` and `library importing` |
| Keywords | `keywords` |

**Note**

Before Robot Framework 4.0 there were also sections for tags and shortcuts. In Robot Framework 4.0 these have been removed in favor of the overview menu. This means that prior linking to shortcuts or tags sections does not work.

**Linking to custom sections**

Robot Framework's [documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-syntax-1) supports custom [section titles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#section-titles), and the titles used in the library or resource file introduction automatically create link targets. The example below illustrates linking both to automatic and custom sections:

*"""Library for Libdoc demonstration purposes.*

*This library does not do anything useful.*

*= My section =*

*We do have a custom section in the documentation, though.*

*"""*

**def** keyword():

*"""Does nothing.*

*See `introduction` for more information and `My section` to test how*

*linking to custom sections works.*

*"""*

**pass**

**Note**

Linking to custom sections works only when using [Robot Framework documentation syntax](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#robot-framework-documentation-syntax).

[**5.1.5   Representing arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-604)

Libdoc shows information about keywords' arguments automatically.

**Included information**

The following information is shown for all keywords regardless are they implemented in libraries or in resource files:

* Argument name. User keyword arguments are shown without the ${} decoration to make arguments look the same regardless where keywords originate from.
* Marker telling is the argument [positional-only](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#positional-only-arguments), [named-only](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-only-arguments), [free positional](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#varargs-library), [free named](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#kwargs-library), or [normal argument](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments) that can be given either by position or by name.
* Possible default value. Shown like = 42.
* Possible type. Shown like <int>. Can be a link to type documentation as explained in the next section.

When referring to arguments in keyword documentation, it is recommended to use [inline code style](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-styles) like ``argument``.

**Automatically listing type documentation**

As mentioned above, Libdoc automatically shows possible type information when listing arguments. If the type is a custom type based on [Enum](https://docs.python.org/library/enum.html#enum.Enum) or [TypedDict](https://docs.python.org/library/typing.html?highlight=typeddict" \l "typing.TypedDict), the type is [automatically converted](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-conversions), or the type has [custom converter](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#custom-argument-converters), also the type itself is listed separately to show more information about it. When these types are used in arguments, the type name also becomes a link to the type information.

All listed data types show possible type documentation as well as what argument types are supported. In addition to that, types based on Enum list available members and types based on TypedDict show the dictionary structure.

**Note**

Automatically listing types based on Enum and TypedDict is new in Robot Framework 4.0. Listing other types is new in Robot Framework 5.0.

[**5.1.6   Libdoc example**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-605)

The following example illustrates how to use the most important [documentation formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting) possibilities, [internal linking](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#internal-linking), and so on. [Click here](https://robotframework.org/robotframework/latest/images/LoggingLibrary.html) to see how the generated documentation looks like.

**class** **LoggingLibrary**:

*"""Library for logging messages.*

*= Table of contents =*

*%TOC%*

*= Usage =*

*This library has several keyword, for example `Log Message`, for logging*

*messages. In reality the library is used only for \_Libdoc\_ demonstration*

*purposes.*

*= Valid log levels =*

*Valid log levels are ``INFO``, ``DEBUG``, and ``TRACE``. The default log*

*level can be set during `importing`.*

*= Examples =*

*Notice how keywords are linked from examples.*

*| `Log Message` | My message | | |*

*| `Log Two Messages` | My message | Second message | level=DEBUG |*

*| `Log Messages` | First message | Second message | Third message |*

*"""*

ROBOT\_LIBRARY\_VERSION = '0.1'

**def** \_\_init\_\_(self, default\_level='INFO'):

*"""The default log level can be given at library import time.*

*See `Valid log levels` section for information about available log*

*levels.*

*Examples:*

*| =Setting= | =Value= | =Value= | =Comment= |*

*| Library | LoggingLibrary | | # Use default level (INFO) |*

*| Library | LoggingLibrary | DEBUG | # Use the given level |*

*"""*

self.default\_level = self.\_verify\_level(default\_level)

**def** \_verify\_level(self, level):

level = level.upper()

**if** level **not** **in** ['INFO', 'DEBUG', 'TRACE']:

**raise** **RuntimeError**("Invalid log level'**%s**'. Valid levels are "

"'INFO', 'DEBUG', and 'TRACE'")

**return** level

**def** log\_message(self, message, level=**None**):

*"""Writes given message to the log file using the specified log level.*

*The message to log and the log level to use are defined using*

*``message`` and ``level`` arguments, respectively.*

*If no log level is given, the default level given during `library*

*importing` is used.*

*"""*

level = self.\_verify\_level(level) **if** level **else** self.default\_level

print("\***%s**\* **%s**" % (level, message))

**def** log\_two\_messages(self, message1, message2, level=**None**):

*"""Writes given messages to the log file using the specified log level.*

*See `Log Message` keyword for more information.*

*"""*

self.log\_message(message1, level)

self.log\_message(message2, level)

**def** log\_messages(self, \*messages):

*"""Logs given messages using the log level set during `importing`.*

*See also `Log Message` and `Log Two Messages`.*

*"""*

**for** msg **in** messages:

self.log\_message(msg)

All [standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#standard-libraries) have documentation generated by Libdoc and their documentation (and source code) act as a more realistic examples.

[**5.2   Test data documentation tool (Testdoc)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-597)

* [5.2.1   General usage](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#general-usage-1)
* [5.2.2   Generating documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generating-documentation-1)

Testdoc is Robot Framework's built-in tool for generating high level documentation based on test cases. The created documentation is in HTML format and it includes name, documentation and other metadata of each test suite and test case, as well as the top-level keywords and their arguments.

[**5.2.1   General usage**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-606)

**Synopsis**

python -m robot.testdoc [options] data\_sources output\_file

**Options**

|  |  |
| --- | --- |
| -T, --title *<title>* | |
|  | Set the title of the generated documentation. Underscores in the title are converted to spaces. The default title is the name of the top level suite. |
| -N, --name *<name>* | |
|  | Override the name of the top level test suite. |
| -D, --doc *<doc>* | |
|  | Override the documentation of the top level test suite. |
| -M, --metadata *<name:value>* | |
|  | Set/override free metadata of the top level test suite. |
| -G, --settag *<tag>* | |
|  | Set given tag(s) to all test cases. |
| -t, --test *<name>* | |
|  | Include tests by name. |
| -s, --suite *<name>* | |
|  | Include suites by name. |
| -i, --include *<tag>* | |
|  | Include tests by tags. |
| -e, --exclude *<tag>* | |
|  | Exclude tests by tags. |
| -A, --argumentfile *<path>* | |
|  | Text file to read more arguments from. Works exactly like [argument files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files) when running tests. |
| -h, --help | Print this help in the console. |

All options except --title have exactly the same semantics as same options have when [executing test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-execution).

[**5.2.2   Generating documentation**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-607)

Data can be given as a single file, directory, or as multiple files and directories. In all these cases, the last argument must be the file where to write the output.

Testdoc can be executed as an installed module like python -m robot.testdoc or as a script like python path/robot/testdoc.py.

Examples:

python -m robot.testdoc my\_test.robot testdoc.html

python path/to/robot/testdoc.py --name "Smoke tests" --include smoke path/to/tests smoke.html

[**5.3   Test data clean-up tool (Tidy)**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-598)

The built-in Tidy tool was deprecated in Robot Framework 4.1 in favor of the new and enhanced external [Robotidy](https://robotidy.readthedocs.io/) tool. It was removed altogether in Robot Framework 5.0.

[**5.4   External tools**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-599)

There are plenty of external tools that can be used with Robot Framework. These tools include plugins for various IDEs and text editors, tools for parallel execution and data-driven testing, plugins for continuous integration systems, and so on.

These tools are developed as separate projects independently from Robot Framework itself. For a list of the available tools see <http://robotframework.org/>.

[**6   Appendices**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-154)

* [6.1   All available settings in test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#all-available-settings-in-test-data)
* [6.2   All command line options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#all-command-line-options)
* [6.3   Documentation formatting](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-formatting)
* [6.4   Time format](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format)
* [6.5   Boolean arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#boolean-arguments)
* [6.6   Evaluating expressions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#evaluating-expressions)
* [6.7   Internal API](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#internal-api)

[**6.1   All available settings in test data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-608)

* [6.1.1   Setting section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-section-1)
* [6.1.2   Test Case section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-section)
* [6.1.3   Keyword section](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-section-1)

[**6.1.1   Setting section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-615)

The Setting section is used to import test libraries, resource files and variable files and to define metadata for test suites and test cases. It can be included in test case files and resource files. Note that in a resource file, a Setting section can only include settings for importing libraries, resources, and variables.

| *Settings available in the Setting section* | |
| --- | --- |
| **Name** | **Description** |
| Library | Used for [importing libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#importing-libraries). |
| Resource | Used for [taking resource files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-resource-files-into-use). |
| Variables | Used for [taking variable files into use](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#taking-variable-files-into-use). |
| Documentation | Used for specifying a [test suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation) or [resource file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-resource-files) documentation. |
| Metadata | Used for setting [free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata). |
| Suite Setup | Used for specifying the [suite setup](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown). |
| Suite Teardown | Used for specifying the [suite teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#suite-setup-and-teardown). |
| Force Tags | Used for specifying forced values for tags when [tagging test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases). |
| Default Tags | Used for specifying default values for tags when [tagging test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases). |
| Test Setup | Used for specifying a default [test setup](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). |
| Test Teardown | Used for specifying a default [test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). |
| Test Template | Used for specifying a default [template keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates) for test cases. |
| Test Timeout | Used for specifying a default [test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout). |
| Task Setup, Task Teardown, Task Template, Task Timeout | Aliases for Test Setup, Test Teardown, Test Template and Test Timeout, respectively, that can be used when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks). |

[**6.1.2   Test Case section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-616)

The settings in the Test Case section are always specific to the test case for which they are defined. Some of these settings override the default values defined in the Settings section.

Exactly same settings are available when [creating tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-tasks) in the Task section.

| *Settings available in the Test Case section* | |
| --- | --- |
| **Name** | **Description** |
| [Documentation] | Used for specifying a [test case documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation). |
| [Tags] | Used for [tagging test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tagging-test-cases). |
| [Setup] | Used for specifying a [test setup](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). |
| [Teardown] | Used for specifying a [test teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-setup-and-teardown). |
| [Template] | Used for specifying a [template keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-templates). |
| [Timeout] | Used for specifying a [test case timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-timeout). |

[**6.1.3   Keyword section**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-617)

Settings in the Keyword section are specific to the user keyword for which they are defined.

| *Settings available in the Keyword section* | |
| --- | --- |
| **Name** | **Description** |
| [Documentation] | Used for specifying a [user keyword documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-documentation). |
| [Tags] | Used for specifying [user keyword tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-tags). |
| [Arguments] | Used for specifying [user keyword arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments). |
| [Return] | Used for specifying [user keyword return values](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-return-values). |
| [Teardown] | Used for specifying [user keyword teardown](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-teardown). |
| [Timeout] | Used for specifying a [user keyword timeout](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-timeout). |

[**6.2   All command line options**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-609)

This appendix lists all the command line options that are available when [executing test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-test-cases-1) and when [post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs). Also environment variables affecting execution and post-processing are listed.

* [6.2.1   Command line options for test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#command-line-options-for-test-execution)
* [6.2.2   Command line options for post-processing outputs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#command-line-options-for-post-processing-outputs)
* [6.2.3   Environment variables for execution and post-processing](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#environment-variables-for-execution-and-post-processing)

[**6.2.1   Command line options for test execution**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-618)

|  |  |
| --- | --- |
| --rpa | Turn on [generic automation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-execution) mode. |
| -F, --extension *<value>* | |
|  | [Parse only these files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#selecting-files-to-parse) when executing a directory. |
| -N, --name *<name>* | |
|  | [Sets the name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-name) of the top-level test suite. |
| -D, --doc *<document>* | |
|  | [Sets the documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-documentation) of the top-level test suite. |
| -M, --metadata *<name:value>* | |
|  | [Sets free metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-free-metadata) for the top level test suite. |
| -G, --settag *<tag>* | |
|  | [Sets the tag(s)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-tags) to all executed test cases. |
| -t, --test *<name>* | |
|  | [Selects the test cases by name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-test-suite-and-test-case-names). |
| --task *<name>* | Alias for --test that can be used when [executing tasks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#executing-tasks). |
| -s, --suite *<name>* | |
|  | [Selects the test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-test-suite-and-test-case-names) by name. |
| -R, --rerunfailed *<file>* | |
|  | [Selects failed tests](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#re-executing-failed-test-cases) from an earlier [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) to be re-executed. |
| -S, --rerunfailedsuites *<file>* | |
|  | [Selects failed test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#re-executing-failed-test-suites) from an earlier [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file) to be re-executed. |
| -i, --include *<tag>* | |
|  | [Selects the test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) by tag. |
| -e, --exclude *<tag>* | |
|  | [Selects the test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) by tag. |
| --skip *<tag>* | Tests having given tag will be [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped). Tag can be a pattern. |
| --skiponfailure *<tag>* | |
|  | Tests having given tag will be [skipped](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#skipped) if they fail. |
| -v, --variable *<name:value>* | |
|  | Sets [individual variables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line). |
| -V, --variablefile *<path:args>* | |
|  | Sets variables using [variable files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files). |
| -d, --outputdir *<dir>* | |
|  | Defines where to [create output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory). |
| -o, --output *<file>* | |
|  | Sets the path to the generated [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file). |
| -l, --log *<file>* | |
|  | Sets the path to the generated [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file). |
| -r, --report *<file>* | |
|  | Sets the path to the generated [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file). |
| -x, --xunit *<file>* | |
|  | Sets the path to the generated [xUnit compatible result file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit-compatible-result-file). |
| -b, --debugfile *<file>* | |
|  | A [debug file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debug-file) that is written during execution. |
| -T, --timestampoutputs | |
|  | [Adds a timestamp](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamping-output-files) to all output files. |
| --splitlog | [Split log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#splitting-logs) into smaller pieces that open in browser transparently. |
| --logtitle *<title>* | |
|  | [Sets a title](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-titles) for the generated test log. |
| --reporttitle *<title>* | |
|  | [Sets a title](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-titles) for the generated test report. |
| --reportbackground *<colors>* | |
|  | [Sets background colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-background-colors) of the generated report. |
| --maxerrorlines *<lines>* | |
|  | Sets the number of [error lines](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-error-message-length-in-reports) shown in report when tests fail. |
| --maxassignlength *<characters>* | |
|  | Sets the number of characters shown in log when [variables are assigned](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatically-logging-assigned-variable-value). |
| -L, --loglevel *<level>* | |
|  | [Sets the threshold level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-log-level) for logging. Optionally the default [visible log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#visible-log-level) can be given separated with a colon (:). |
| --suitestatlevel *<level>* | |
|  | Defines how many [levels to show](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-displayed-suite-statistics) in the *Statistics by Suite* table in outputs. |
| --tagstatinclude *<tag>* | |
|  | [Includes only these tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#including-and-excluding-tag-statistics) in the *Statistics by Tag* table. |
| --tagstatexclude *<tag>* | |
|  | [Excludes these tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#including-and-excluding-tag-statistics) from the *Statistics by Tag* table. |
| --tagstatcombine *<tags:title>* | |
|  | Creates [combined statistics based on tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generating-combined-tag-statistics). |
| --tagdoc *<pattern:doc>* | |
|  | Adds [documentation to the specified tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#adding-documentation-to-tags). |
| --tagstatlink *<pattern:link:title>* | |
|  | Adds [external links](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-links-from-tag-names) to the *Statistics by Tag* table. |
| --expandkeywords *<name:pattern|tag:pattern>* | |
|  | Automatically [expand keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatically-expanding-keywords) in the generated log file. |
| --removekeywords *<all|passed|name:pattern|tag:pattern|for|while|wuks>* | |
|  | [Removes keyword data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) from the generated log file. |
| --flattenkeywords *<for|while|iteration|name:pattern|tag:pattern>* | |
|  | [Flattens keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) in the generated log file. |
| --listener *<name:args>* | |
|  | [Sets a listener](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-listeners) for monitoring test execution. |
| --nostatusrc | Sets the [return code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes) to zero regardless of failures in test cases. Error codes are returned normally. |
| --runemptysuite | |
|  | Executes tests also if the selected [test suites are empty](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#when-no-tests-match-selection). |
| --dryrun | In the [dry run](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dry-run) mode tests are run without executing keywords originating from test libraries. Useful for validating test data syntax. |
| -X, --exitonfailure | |
|  | [Stops test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-when-first-test-case-fails) if any test fails. |
| --exitonerror | [Stops test execution](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#stopping-on-parsing-or-execution-error) if any error occurs when parsing test data, importing libraries, and so on. |
| --skipteardownonexit | |
|  | [Skips teardowns](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#handling-teardowns) if test execution is prematurely stopped. |
| --prerunmodifier *<name:args>* | |
|  | Activate [programmatic modification of test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-test-data). |
| --prerebotmodifier *<name:args>* | |
|  | Activate [programmatic modification of results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-results). |
| --randomize *<all|suites|tests|none>* | |
|  | [Randomizes](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#randomizing-execution-order) test execution order. |
| --console *<verbose|dotted|quiet|none>* | |
|  | [Console output type](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-output-type). |
| --dotted | Shortcut for --console dotted. |
| --quiet | Shortcut for --console quiet. |
| -W, --consolewidth *<width>* | |
|  | [Sets the width](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-width) of the console output. |
| -C, --consolecolors *<auto|on|ansi|off>* | |
|  | [Specifies are colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-colors) used on the console. |
| -K, --consolemarkers *<auto|on|off>* | |
|  | Show [markers on the console](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-markers) when top level keywords in a test case end. |
| -P, --pythonpath *<path>* | |
|  | Additional locations to add to the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path). |
| -A, --argumentfile *<path>* | |
|  | A text file to [read more arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files) from. |
| -h, --help | Prints [usage instructions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-help-and-version-information). |
| --version | Prints the [version information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-help-and-version-information). |

[**6.2.2   Command line options for post-processing outputs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-619)

|  |  |
| --- | --- |
| --rpa | Turn on [generic automation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#task-execution) mode. |
| -R, --merge | Changes result combining behavior to [merging](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#merging-outputs). |
| -N, --name *<name>* | |
|  | [Sets the name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-name) of the top level test suite. |
| -D, --doc *<document>* | |
|  | [Sets the documentation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-the-documentation) of the top-level test suite. |
| -M, --metadata *<name:value>* | |
|  | [Sets free metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-free-metadata) for the top-level test suite. |
| -G, --settag *<tag>* | |
|  | [Sets the tag(s)](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-tags) to all processed test cases. |
| -t, --test *<name>* | |
|  | [Selects the test cases by name](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-test-suite-and-test-case-names). |
| --task *<name>* | Alias for --test. |
| -s, --suite *<name>* | |
|  | [Selects the test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-test-suite-and-test-case-names) by name. |
| -i, --include *<tag>* | |
|  | [Selects the test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) by tag. |
| -e, --exclude *<tag>* | |
|  | [Selects the test cases](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#by-tag-names) by tag. |
| -d, --outputdir *<dir>* | |
|  | Defines where to [create output files](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-directory). |
| -o, --output *<file>* | |
|  | Sets the path to the generated [output file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#output-file). |
| -l, --log *<file>* | |
|  | Sets the path to the generated [log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-file). |
| -r, --report *<file>* | |
|  | Sets the path to the generated [report file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#report-file). |
| -x, --xunit *<file>* | |
|  | Sets the path to the generated [xUnit compatible result file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html" \l "xunit-compatible-result-file). |
| -T, --timestampoutputs | |
|  | [Adds a timestamp](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timestamping-output-files) to all output files. |
| --splitlog | [Split log file](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#splitting-logs) into smaller pieces that open in browser transparently. |
| --logtitle *<title>* | |
|  | [Sets a title](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-titles) for the generated test log. |
| --reporttitle *<title>* | |
|  | [Sets a title](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-titles) for the generated test report. |
| --reportbackground *<colors>* | |
|  | [Sets background colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-background-colors) of the generated report. |
| -L, --loglevel *<level>* | |
|  | [Sets the threshold level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-log-level) to select log messages. Optionally the default [visible log level](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#visible-log-level) can be given separated with a colon (:). |
| --suitestatlevel *<level>* | |
|  | Defines how many [levels to show](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-displayed-suite-statistics) in the *Statistics by Suite* table in outputs. |
| --tagstatinclude *<tag>* | |
|  | [Includes only these tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#including-and-excluding-tag-statistics) in the *Statistics by Tag* table. |
| --tagstatexclude *<tag>* | |
|  | [Excludes these tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#including-and-excluding-tag-statistics) from the *Statistics by Tag* table. |
| --tagstatcombine *<tags:title>* | |
|  | Creates [combined statistics based on tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#generating-combined-tag-statistics). |
| --tagdoc *<pattern:doc>* | |
|  | Adds [documentation to the specified tags](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#adding-documentation-to-tags). |
| --tagstatlink *<pattern:link:title>* | |
|  | Adds [external links](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-links-from-tag-names) to the *Statistics by Tag* table. |
| --expandkeywords *<name:pattern|tag:pattern>* | |
|  | Automatically [expand keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#automatically-expanding-keywords) in the generated log file. |
| --removekeywords *<all|passed|name:pattern|tag:pattern|for|wuks>* | |
|  | [Removes keyword data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) from the generated outputs. |
| --flattenkeywords *<for|foritem|name:pattern|tag:pattern>* | |
|  | [Flattens keywords](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#removing-and-flattening-keywords) in the generated outputs. |
| --starttime *<timestamp>* | |
|  | Sets the [starting time](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-start-and-end-time-of-execution) of test execution when creating reports. |
| --endtime *<timestamp>* | |
|  | Sets the [ending time](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-start-and-end-time-of-execution) of test execution when creating reports. |
| --nostatusrc | Sets the [return code](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#return-codes) to zero regardless of failures in test cases. Error codes are returned normally. |
| --processemptysuite | |
|  | Processes output files even if files contain [empty test suites](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#when-no-tests-match-selection). |
| --prerebotmodifier *<name:args>* | |
|  | Activate [programmatic modification of results](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#programmatic-modification-of-results). |
| -C, --consolecolors *<auto|on|ansi|off>* | |
|  | [Specifies are colors](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#console-colors) used on the console. |
| -P, --pythonpath *<path>* | |
|  | Additional locations to add to the [module search path](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#module-search-path). |
| -A, --argumentfile *<path>* | |
|  | A text file to [read more arguments](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files) from. |
| -h, --help | Prints [usage instructions](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-help-and-version-information). |
| --version | Prints the [version information](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-help-and-version-information). |

[**6.2.3   Environment variables for execution and post-processing**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-620)

ROBOT\_OPTIONS and REBOT\_OPTIONS

Space separated list of default options to be placed [in front of any explicit options](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#robot-options-and-rebot-options-environment-variables) on the command line.

ROBOT\_SYSLOG\_FILE

Path to a [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) file where Robot Framework writes internal information about parsing test case files and running tests.

ROBOT\_SYSLOG\_LEVEL

Log level to use when writing to the [syslog](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#system-log) file.

ROBOT\_INTERNAL\_TRACES

When set to any non-empty value, Robot Framework's internal methods are included in [error tracebacks](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#debugging-problems).

[**6.3   Documentation formatting**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-610)

It is possible to use simple HTML formatting with [test suite](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-suite-name-and-documentation), [test case](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-case-name-and-documentation) and [user keyword](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-documentation) documentation and [free test suite metadata](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#free-test-suite-metadata) in the test data, as well as when [documenting test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documenting-libraries). The formatting is similar to the style used in most wikis, and it is designed to be understandable both as plain text and after the HTML transformation.

* [6.3.1   Representing newlines](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#representing-newlines)
  + [Newlines in test data](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#newlines-in-test-data)
  + [Documentation in test libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#documentation-in-test-libraries)
* [6.3.2   Paragraphs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#paragraphs)
* [6.3.3   Inline styles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-styles)
* [6.3.4   URLs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#urls)
* [6.3.5   Custom links and images](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#custom-links-and-images)
  + [Link with text content](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#link-with-text-content)
  + [Link with image content](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#link-with-image-content)
  + [Image with title text](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#image-with-title-text)
* [6.3.6   Section titles](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#section-titles)
* [6.3.7   Tables](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#tables)
* [6.3.8   Lists](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#lists)
* [6.3.9   Preformatted text](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#preformatted-text)
* [6.3.10   Horizontal ruler](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#horizontal-ruler)

[**6.3.1   Representing newlines**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-621)

[**Newlines in test data**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-622)

When documenting test suites, test cases and keywords or adding metadata to test suites, newlines can be added manually using \n [escape sequence](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escape-sequence).

**\*\*\* Settings \*\*\***

**Documentation** First line.\n\nSecond paragraph. This time\nwith multiple lines.

**Metadata** Example list - first item\n- second item\n- third

**Note**

As explained in the [Paragraphs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#paragraphs) section below, the single newline in Second paragraph, this time\nwith multiple lines. does not actually affect how that paragraph is rendered. Newlines are needed when creating [lists](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#lists) or other such constructs, though.

Adding newlines manually to a long documentation takes some effort and extra characters also make the documentation harder to read. This can be avoided, though, as newlines are inserted automatically between [continued documentation and metadata lines](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#dividing-data-to-several-rows). In practice this means that the above example could be written also as follows.

**\*\*\* Settings \*\*\***

**Documentation**

... First line.

...

... Second paragraph. This time

... with multiple lines.

**Metadata** Example list

... - first item

... - second item

... - third

No automatic newline is added if a line already ends with a literal newline or if it ends with an [escaping backslash](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping). If documentation or metadata is defined in multiple columns, cells in a same row are concatenated together with a space. Different ways to split documentation are illustrated in the examples below where all test cases end up having the same two line documentation.

**\*\*\* Test Cases \*\*\***

**Example 1**

[**Documentation**] First line\n Second line in multiple parts

No Operation

**Example 2**

[**Documentation**] First line

... Second line in multiple parts

No Operation

**Example 3**

[**Documentation**] First line\n

... Second line in\

... multiple parts

No Operation

[**Documentation in test libraries**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-623)

With library documentations normal newlines are enough, and for example the following keyword documentation would create same end result as the test suite documentation in the previous section.

**def** example\_keyword():

*"""First line.*

*Second paragraph, this time*

*with multiple lines.*

*"""*

**pass**

[**6.3.2   Paragraphs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-624)

All regular text in the formatted HTML documentation is represented as paragraphs. In practice, lines separated by a single newline will be combined in a paragraph regardless whether the newline is added manually or automatically. Multiple paragraphs can be separated with an empty line (i.e. two newlines) and also tables, lists, and other specially formatted blocks discussed in subsequent sections end a paragraph.

For example, the following test suite or resource file documentation:

**\*\*\* Settings \*\*\***

**Documentation**

... First paragraph has only one line.

...

... Second paragraph, this time created

... with multiple lines.

will be formatted in HTML as:

First paragraph has only one line.

Second paragraph, this time created with multiple lines.

[**6.3.3   Inline styles**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-625)

The documentation syntax supports inline styles **bold**, *italic* and code. Bold text can be created by having an asterisk before and after the selected word or words, for example \*this is bold\*. Italic style works similarly, but the special character to use is an underscore, for example, \_italic\_. It is also possible to have bold italic with the syntax \_\*bold italic\*\_.

The code style is created using double backticks like ``code``. The result is monospaced text with light gray background.

Asterisks, underscores or double backticks alone, or in the middle of a word, do not start formatting, but punctuation characters before or after them are allowed. When multiple lines form a [paragraph](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#paragraphs), all inline styles can span over multiple lines.

|  |  |
| --- | --- |
| *Inline style examples* | |
| **Unformatted** | **Formatted** |
| \*bold\* | **bold** |
| \_italic\_ | *italic* |
| \_\*bold italic\*\_ | ***bold italic*** |
| ``code`` | code |
| \*bold\*, then \_italic\_ and finally ``some code`` | **bold**, then *italic* and finally some code |
| This is \*bold\n on multiple\n lines\*. | This is **bold** **on multiple** **lines**. |

[**6.3.4   URLs**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-626)

All strings that look like URLs are automatically converted into clickable links. Additionally, URLs that end with extension *.jpg*, *.jpeg*, *.png*, *.gif*, *.bmp* or *.svg* (case-insensitive) will automatically create images. For example, URLs like http://example.com are turned into links, and http:///host/image.jpg and file:///path/chart.png into images.

The automatic conversion of URLs to links is applied to all the data in logs and reports, but creating images is done only for test suite, test case and keyword documentation, and for test suite metadata.

**Note**

*.svg* image support is new in Robot Framework 3.2.

[**6.3.5   Custom links and images**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-627)

It is possible to create custom links and embed images using special syntax [link|content]. This creates a link or image depending are link and content images. They are considered images if they have the same image extensions that are special with [URLs](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#urls) or start with data:image/. The surrounding square brackets and the pipe character between the parts are mandatory in all cases.

**Note**

Support for the data:image/ prefix is new in Robot Framework 3.2.

[**Link with text content**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-628)

If neither link nor content is an image, the end result is a normal link where link is the link target and content the visible text:

[file.html|this file] -> <a href="file.html">this file</a>

[http://host|that host] -> <a href="http://host">that host</a>

[**Link with image content**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-629)

If content is an image, you get a link where the link content is an image. Link target is created by link and it can be either text or image:

[robot.html|robot.png] -> <a href="robot.html"><img src="robot.png"></a>

[robot.html|data:image/png;base64,oooxxx=] -> <a href="robot.html"><img src="data:image/png;base64,oooxxx="></a>

[image.jpg|thumb.jpg] -> <a href="image.jpg"><img src="thumb.jpg"></a>

[**Image with title text**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-630)

If link is an image but content is not, the syntax creates an image where the content is the title text shown when mouse is over the image:

[robot.jpeg|Robot rocks!] -> <img src="robot.jpeg" title="Robot rocks!">

[data:image/png;base64,oooxxx=|Robot rocks!] -> <img src="data:image/png;base64,oooxxx=" title="Robot rocks!">

[**6.3.6   Section titles**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-631)

If documentation gets longer, it is often a good idea to split it into sections. It is possible to separate sections with titles using syntax = My Title =, where the number of equal signs denotes the level of the title:

= First section =

== Subsection ==

Some text.

== Second subsection ==

More text.

= Second section =

You probably got the idea.

Notice that only three title levels are supported and that spaces between equal signs and the title text are mandatory.

[**6.3.7   Tables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-632)

Tables are created using pipe characters with spaces around them as column separators and newlines as row separators. Header cells can be created by surrounding the cell content with equal signs and optional spaces like = Header = or =Header=. Tables cells can also contain links and formatting such as bold and italic:

| =A= | =B= | = C = |

| \_1\_ | Hello | world! |

| \_2\_ | Hi |

The created table always has a thin border and normal text is left-aligned. Text in header cells is bold and centered. Empty cells are automatically added to make rows equally long. For example, the above example would be formatted like this in HTML:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| *1* | Hello | world |
| *2* | Hi |  |

[**6.3.8   Lists**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-633)

Lists are created by starting a line with a hyphen and space ('- '). List items can be split into multiple lines by indenting continuing lines with one or more spaces. A line that does not start with '- ' and is not indented ends the list:

Example:

- a list item

- second list item

is continued

This is outside the list.

The above documentation is formatted like this in HTML:

Example:

* a list item
* second list item is continued

This is outside the list.

[**6.3.9   Preformatted text**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-634)

It is possible to embed blocks of preformatted text in the documentation. Preformatted block is created by starting lines with '| ', one space being mandatory after the pipe character except on otherwise empty lines. The starting '| ' sequence will be removed from the resulting HTML, but all other whitespace is preserved.

In the following documentation, the two middle lines form a preformatted block when converted to HTML:

Doc before block:

| inside block

| some additional whitespace

After block.

The above documentation is formatted like this:

Doc before block:

inside block

some additional whitespace

After block.

When documenting suites, tests or keywords in Robot Framework test data, having multiple spaces requires [escaping](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#escaping) with a backslash to prevent ignoring spaces. The example above would thus be written like this:

Doc before block:

| inside block

| \ \ \ some \ \ additional whitespace

After block.

[**6.3.10   Horizontal ruler**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-635)

Horizontal rulers (the <hr> tag) make it possible to separate larger sections from each others, and they can be created by having three or more hyphens alone on a line:

Some text here.

---

More text...

The above documentation is formatted like this:

Some text here.

More text...

[**6.4   Time format**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-611)

Robot Framework has its own time format that is both flexible to use and easy to understand. It is used by several keywords (for example, [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords *Sleep* and *Wait Until Keyword Succeeds*), [DateTime](https://robotframework.org/robotframework/latest/libraries/DateTime.html) library, and [timeouts](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#timeouts).

[**6.4.1   Time as number**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-175)

The time can always be given as a plain number, in which case it is interpreted to be seconds. Both integers and floating point numbers work, and it is possible to use either real numbers or strings containing numerical values.

**Note**

In some contexts plain numbers can be interpreted otherwise as times. For example, with [WHILE loop limit](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-while-loop-iterations) integers denote the maximum iteration count.

[**6.4.2   Time as time string**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-176)

Representing the time as a time string means using a format such as 2 minutes 42 seconds, which is normally easier to understand than just having the value as seconds. It is, for example, not so easy to understand how long a time 4200 is in seconds, but 1 hour 10 minutes is clear immediately.

The basic idea of this format is having first a number and then a text specifying what time that number represents. Numbers can be either integers or floating point numbers, the whole format is case and space insensitive, and it is possible to add - prefix to specify negative times. The available time specifiers are:

* days, day, d
* hours, hour, h
* minutes, minute, mins, min, m
* seconds, second, secs, sec, s
* milliseconds, millisecond, millis, ms

Examples:

1 min 30 secs

1.5 minutes

90 s

1 day 2 hours 3 minutes 4 seconds 5 milliseconds

1d 2h 3m 4s 5ms

- 10 seconds

[**6.4.3   Time as "timer" string**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-177)

Time can also be given in timer like format hh:mm:ss.mil. In this format both hour and millisecond parts are optional, leading and trailing zeros can be left out when they are not meaningful, and negative times can be represented by adding the - prefix. For example, following timer and time string values are identical:

| *Timer and time string examples* | |
| --- | --- |
| **Timer** | **Time string** |
| 00:00:01 | 1 second |
| 01:02:03 | 1 hour 2 minutes 3 seconds |
| 1:00:00 | 1 hour |
| 100:00:00 | 100 hours |
| 00:02 | 2 seconds |
| 42:00 | 42 minutes |
| 00:01:02.003 | 1 minute 2 seconds 3 milliseconds |
| 00:01.5 | 1.5 seconds |
| -01:02.345 | - 1 minute 2 seconds 345 milliseconds |

[**6.5   Boolean arguments**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-612)

Many keywords in Robot Framework [standard libraries](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#standard-libraries) 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. 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](http://docs.python.org/library/stdtypes.html#truth-value-testing).

**\*\*\* Keywords \*\*\***

**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 *# Special false string with this keyword.*

**Note**

Considering OFF and 0 false is new in Robot Framework 3.1.

[**6.6   Evaluating expressions**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-613)

Constructs such as [IF/ELSE structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else-structures), [WHILE loops](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#while-loops) and [inline Python evaluation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-python-evaluation) as well as several [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords accept an expression that is evaluated in Python:

**\*\*\* Test Cases \*\*\***

**If expression**

IF ${x} > 0

Log to console ${x} is positive

ELSE

Log to console ${x} is negative

END

**Inline Python evaluation**

Log to console ${x} is ${{'positive' if ${x} > 0 else 'negative'}}

**Evaluate keyword**

${type} = Evaluate 'positive' if ${x} > 0 else 'negative'

Log to console ${x} is ${type}

**Should Be True keyword**

Should Be True ${x} > 0

This section explains how the expression is evaluated and how variables in the expression are handled. Notice that instead of creating complicated expressions, it is often better to move the logic into a [test library](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries). That typically eases maintenance and also enhances execution speed.

[**6.6.1   Evaluation namespace**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-180)

Expressions are evaluated using Python's [eval](http://docs.python.org/library/functions.html#eval) function so that normal Python constructs like '${x}' == 'expected', ${x} > 0 and '${x}'.upper() not in ('FAIL', 'BAD') can be used and all builtin functions like len() and int() are available. In addition to that, all unrecognized Python 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.

The following examples demonstrate using Python builtins as well as modules using the [inline Python evaluation](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#inline-python-evaluation) syntax, but same expressions would also work with [IF/ELSE structures](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#if-else-structures) and [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keywords without the need to use the ${{}} decoration around the expression:

**\*\*\* Variables \*\*\***

${VAR} 123

**\*\*\* Test Cases \*\*\***

**Python syntax**

Should Be True ${{'${VAR}' == '123'}}

Should Be True ${{'${VAR}'.startswith('x') or '${VAR}' in '012345'}}

**Python builtins**

Should Be Equal ${{len('${VAR}')}} ${3}

Should Be Equal ${{int('${VAR}')}} ${123}

**Access modules**

Should Be Equal ${{os.sep}} ${/}

Should Be Equal ${{round(math.pi, 2)}} ${3.14}

Should Start With ${{robot.\_\_version\_\_}} 4.

A limitation of using modules is that nested modules like rootmod.submod can only be used if the root module automatically imports the submodule. That is not always the case and using such modules is not possible. An concrete example that is relevant in the automation context is the selenium module that is implemented, at least at the time of this writing, so that just importing selenium does not import the selenium.webdriver submodule. Another limitation is that modules cannot be used in the expression part of a list comprehension when using Python 3. A workaround to both of these problems is using the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) keyword *Evaluate* that accepts modules to be imported and added to the evaluation namespace as an argument:

**\*\*\* Test Cases \*\*\***

**Does not work due to nested module structure**

Log ${{selenium.webdriver.ChromeOptions()}}

**Evaluate keyword with nested module**

${options} = Evaluate selenium.webdriver.ChromeOptions() modules=selenium.webdriver

Log ${options}

**Does not work due to list comprehension**

Log ${{[json.loads(item) for item in ('1', '"b"')]}}

**Evaluate keyword with list comprehension**

${items} = Evaluate [json.loads(item) for item in ('1', '"b"')] modules=json

Log ${items}

The *Evaluate* keyword also supports custom evaluation namespaces if further customization is needed. See its documentation in the [BuiltIn](https://robotframework.org/robotframework/latest/libraries/BuiltIn.html) library for more details.

[**6.6.2   Using variables**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-181)

**Normal ${variable} syntax**

When a variable is used in the expression 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. For example, if we have a return code as an integer in variable ${rc}, using something like ${rc} > 0 is fine.

With other objects the behavior depends on the string representation. Most importantly, strings must always be quoted either with single or double quotes like '${x}', and if they can contain newlines, they must be triple-quoted like '''${x}'''. Strings containing quotes themselves cause additional problems, but triple-quoting typically handles them. Also the backslash character \ is problematic, but can be handled by using Python's raw-string notation like r'${path}'.

**\*\*\* Test Cases \*\*\***

**Using normal variable syntax**

Should Be True ${rc} > 0

IF '${status}'.upper() == 'PASS'

Log Passed

END

IF 'FAIL' in r'''${output}'''

Log Output contains FAIL

END

**Special $variable syntax**

Quoting strings is not that convenient, but there are cases where replacing the variable with its string representation causes even bigger problems. For example, if the variable value can be either a string or Python None, quoting like '${var}' is needed because otherwise strings do not work, but then None is interpreted to be a string as well. Luckily there is an easy solution to these problems discussed in this section.

Actual variables values are available in the evaluation namespace and can be accessed using special variable syntax without the curly braces like $variable. Such variables should never be quoted, not even if they contain strings.

Compare this these examples with the example in the previous section:

**\*\*\* Test Cases \*\*\***

**Using special variable syntax**

Should Be True $rc > 0

IF $status.upper() == 'PASS'

Log Passed

END

IF 'FAIL' in $output

Log Output contains FAIL

END

**Only possible using special variable syntax**

Should Be True $example is not None

Should Be True len($result) > 1 and $result[1] == 'OK'

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. Moving such logic to test libraries is typically a good idea anyway.

[**6.7   Internal API**](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#toc-entry-614)

[API documentation](http://robot-framework.readthedocs.org/) is hosted separately at the excellent [Read the Docs](http://readthedocs.org/) service. If you are unsure how to use certain API or is using them forward compatible, please send a question to [mailing list](https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#mailing-lists).