Robot Framework User Guide

Version 3.1.1

Copyright © 2008-2015 Nokia Networks
Copyright © 2016- Robot Framework Foundation
Licensed under the Creative Commons Attribution 3.0 Unported license

Table of Contents

1 Getting started

1.1 Introduction

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.

Robot Framework is open source software released under the Apache License 2.0. Its development is sponsored by the Robot Framework Foundation.

Note

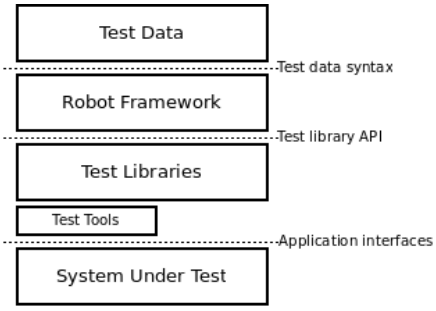
The official RRA support was added in Robot Framework 3.1. This User Guide still talks mainly about creating tests, test data, and test libraries, but same concepts apply also when creating tasks.

1.1.1 Why Robot Framework?

- Enables easy-to-use tabular syntax for creating test cases in a uniform way.
- Provides ability to create reusable higher-level keywords from the existing keywords.
- Provides easy-to-read result reports and logs in HTML format.
- Is platform and application independent.
- Provides a simple library API for creating customized test libraries which can be implemented natively with either Python or Java.
- Provides a command line interface and XML based output files for integration into existing build infrastructure (continuous integration systems).
- Provides support for Selenium for web testing, Java GUI testing, running processes, Telnet, SSH, and so on.
- Supports creating data-driven test cases.
- Has built-in support for variables, practical particularly for testing in different environments.
- Provides tagging to categorize and select test cases to be executed.
- Enables easy integration with source control: test suites are just files and directories that can be versioned with the production code.
- Provides test-case and test-suite -level setup and teardown.
- The modular architecture supports creating tests even for applications with several diverse interfaces.

1.1.2 High-level architecture

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

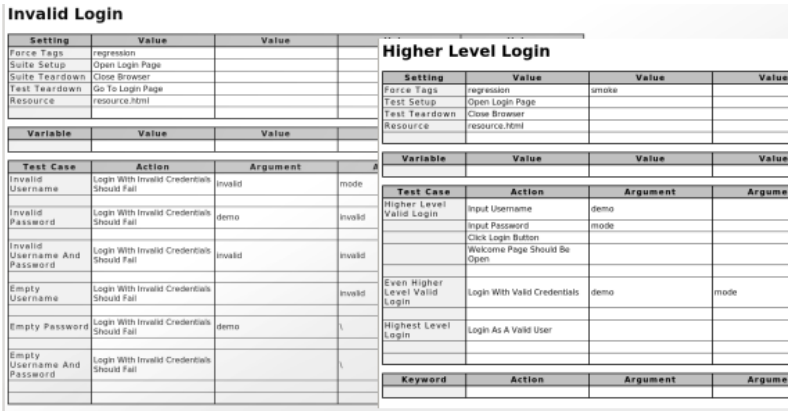


Robot Framework architecture

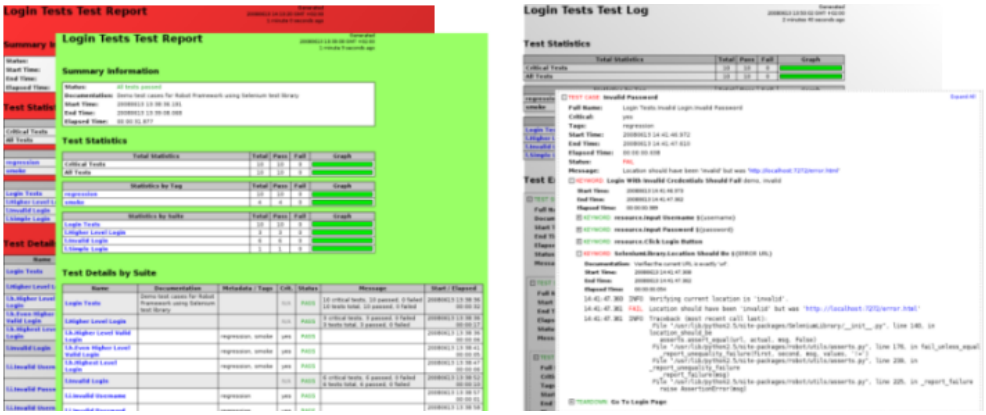
The test data is in simple, easy-to-edit tabular format. When Robot Framework is started, it processes the data, executes test cases and generates logs and reports. The core framework does not know anything about the target under test, and the interaction with it is handled by libraries. Libraries can either use application interfaces directly or use lower level test tools as drivers.

1.1.3 Screenshots

Following screenshots show examples of the test data and created reports and logs.



Test case files



Reports and logs

1.1.4 Getting more information

Project pages

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

Mailing lists

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.

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
: 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
: Discussion about Robot Framework development.

1.2 Copyright and license

Robot Framework is open source software provided under the Apache License 2.0. Robot Framework documentation such as this User Guide use the Creative Commons Attribution 3.0 Unported 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

These instructions cover installing and uninstalling Robot Framework and its preconditions on different operating systems. If you already have pip installed, it is enough to run:

```
pip install robotframework
```


1.3.1 Introduction

Robot Framework is implemented with Python and supports also Jython (JVM), IronPython (.NET) and PyPy. Before installing the framework, an obvious precondition is installing at least one of these interpreters.

Different ways to install Robot Framework itself are listed below and explained more thoroughly in the subsequent sections.

Installing with pip
: Using pip is the recommended way to install Robot Framework. As the standard Python package manager it is included in the latest Python, Jython and IronPython versions. If you already have pip available, you can simply execute:

  ```
  pip install robotframework
  ```

Installing from source
: This approach works regardless the operating system and the Python interpreter used. You can get the source code either by downloading a source distribution from PyPI and extracting it, or by cloning the GitHub repository .

Standalone JAR distribution
: If running tests with Jython is enough, the easiest approach is downloading the standalone `robotframework-<version>.jar` from Maven central. The JAR distribution contains both Jython and Robot Framework and thus only requires having Java installed.

Manual installation
: If you have special needs and nothing else works, you can always do a custom manual

installation.

> **Note**
>
> Prior to Robot Framework 3.0, there were also separate Windows installers for 32bit and 64bit Python versions. Because Python 2.7.9 and newer contain pip on Windows and Python 3 would have needed two more installers, it was decided that Windows installers are not created anymore. The recommend installation approach also on Windows is using pip.

1.3.2 Preconditions

Robot Framework is supported on Python (both Python 2 and Python 3), Jython (JVM) and IronPython (.NET) and PyPy. The interpreter you want to use should be installed before installing the framework itself.

Which interpreter to use depends on the needed test libraries and test environment in general. Some libraries use tools or modules that only work with Python, while others may use Java tools that require Jython or need .NET and thus IronPython. There are also many tools and libraries that run fine with all interpreters.

If you do not have special needs or just want to try out the framework, it is recommended to use Python. It is the most mature implementation, considerably faster than Jython or IronPython (especially start-up time is faster), and also readily available on most UNIX-like operating systems. Another good alternative is using the standalone JAR distribution that only has Java as a precondition.

Python 2 vs Python 3

Python 2 and Python 3 are mostly the same language, but they are not fully compatible with each others. The main difference is that in Python 3 all strings are Unicode while in Python 2 strings are bytes by default, but there are also several other backwards incompatible changes. The last Python 2 release is Python 2.7 that was released in 2010 and will be supported until 2020. See Should I use Python 2 or 3? for more information about the differences, which version to use, how to write code that works with both versions, and so on.

Robot Framework 3.0 is the first Robot Framework version to support Python 3. It supports also Python 2, and the plan is to continue Python 2 support as long as Python 2 itself is officially supported. We hope that authors of the libraries and tools in the wider Robot Framework ecosystem also start looking at Python 3 support now that the core framework supports it.

Python installation

On most UNIX-like systems such as Linux and OS X you have Python installed by default. If you are on Windows or otherwise need to install Python yourself, a good place to start is http://python.org. There you can download a suitable installer and get more information about the installation process and Python in general.

Robot Framework 3.0 supports Python 2.6, 2.7, 3.3 and newer, but the plan is to drop Python 2.6 and 3.3 support in RF 3.1. If you need to use older versions, Robot Framework 2.5-2.8 support Python 2.5 and Robot Framework 2.0-2.1 support Python 2.3 and 2.4.

After installing Python, you probably still want to configure PATH to make Python itself as well as the `robot` and `rebot` runner scripts executable on the command line.

> **Tip**
>
> Latest Python Windows installers allow setting `PATH` as part of the installation. This is disabled by default, but `Add python.exe to Path` can be enabled on the `Customize Python` screen.

Jython installation

Using test libraries implemented with Java or that use Java tools internally requires running Robot Framework on Jython, which in turn requires Java Runtime Environment (JRE) or Java Development Kit (JDK). Installing either of these Java distributions is out of the scope of these instructions, but you can find more information, for example, from http://java.com.

Installing Jython is a fairly easy procedure, and the first step is getting an installer from http://jython.org. The installer is an executable JAR package, which you can run from the command line like `java -jar jython_installer-<version>.jar`. Depending on the system configuration, it may also be possible to just double-click the installer.

Robot Framework 3.0 supports Jython 2.7 which requires Java 7 or newer. If older Jython or Java versions are needed, Robot Framework 2.5-2.8 support Jython 2.5 (requires Java 5 or newer) and Robot Framework 2.0-2.1 support Jython 2.2.

After installing Jython, you probably still want to configure PATH to make Jython itself as well as the `robot` and `rebot` runner scripts executable on the command line.

IronPython installation

IronPython allows running Robot Framework on the .NET platform and interacting with C# and other .NET languages and APIs. Only IronPython 2.7 is supported in general and IronPython 2.7.9 or newer is highly recommended.

If not using IronPython 2.7.9 or newer and Robot Framework 3.1 or newer, an additional requirement is installing ElementTree module 1.2.7 preview release. This is required because the ElementTree module distributed with older IronPython versions was broken. Once you have pip activated for IronPython, you can easily install ElementTree using this command:

```
ipy -m pip install http://effbot.org/media/downloads/elementtree-1.2.7-20070827-preview.zip
```

Alternatively you can download the zip package, extract it, and install it by running `ipy setup.py install` on the command prompt in the created directory.

After installing IronPython, you probably still want to configure PATH to make IronPython itself as well as the `robot` and `rebot` runner scripts executable on the command line.

PyPy installation

PyPy is an alternative implementation of the Python language with both Python 2 and Python 3 compatible versions available. 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. After installation you probably still want to configure PATH to make PyPy itself as well as the `robot` and `rebot` runner scripts executable on the command line.

Configuring `PATH`

The `PATH` environment variable lists locations where commands executed in a system are searched from. To make using Robot Framework easier from the command prompt, it is recommended to add the locations where the runner scripts are installed into the `PATH`. It is also often useful to have the interpreter itself in the `PATH` to make executing it easy.

When using Python on UNIX-like machines both Python itself and scripts installed with should be automatically in the `PATH` and no extra actions needed. On Windows and with other interpreters the `PATH` must be configured separately.

> **Tip**
>
> Latest Python Windows installers allow setting `PATH` as part of the installation. This is disabled by default, but `Add python.exe to Path` can be enabled on the `Customize Python` screen. It will add both the Python installation directory and the *Scripts* directory to the `PATH`.

What directories to add to `PATH`

What directories you need to add to the `PATH` depends on the interpreter and the operating system. The first location is the installation directory of the interpreter (e.g. *C:\Python27*) and the other is the location where scripts are installed with that interpreter. Both Python and IronPython install scripts to *Scripts* directory under the installation directory on Windows (e.g. *C:\Python27\Scripts*) and Jython uses *bin* directory regardless the operating system (e.g. *C:\jython2.7.0\bin*).

Notice that the *Scripts* and *bin* directories may not be created as part of the interpreter installation, but only later when Robot Framework or some other third party module is installed.

Setting `PATH` on Windows

On Windows you can configure `PATH` by following the steps below. Notice that the exact setting names may be different on different Windows versions, but the basic approach should still be the same.

1. Open `Control Panel > System > Advanced > Environment Variables`. There are `User variables` and `System variables`, and the difference between them is that user variables affect only the current users, whereas system variables affect all users.
2. To edit an existing `PATH` value, select `Edit` and add `;<InstallationDir>;<ScriptsDir>` at the end of the value (e.g. `;C:\Python27;C:\Python27\Scripts`). Note that the semicolons (`;`) are important as they separate the different entries. To add a new `PATH` value, select `New` and set both the name and the value, this time without the leading semicolon.

3. Exit the dialog with `Ok` to save the changes.
4. Start a new command prompt for the changes to take effect.

Notice that if you have multiple Python versions installed, the executed `robot` or `rebot` runner script will always use the one that is *first* in the `PATH` regardless under what Python version that script is installed. To avoid that, you can always execute the installed robot module directly like `C:\Python27\python.exe -m robot`.

Notice also that you should not add quotes around directories you add into the `PATH` (e.g. `"C:\Python27\Scripts"`). Quotes can cause problems with Python programs and they are not needed in this context even if the directory path would contain spaces.

Setting `PATH` on UNIX-like systems

On UNIX-like systems you typically need to edit either some system wide or user specific configuration file. Which file to edit and how depends on the system, and you need to consult your operating system documentation for more details.

Setting `https_proxy`

If you are installing with pip and are behind a proxy, you need to set the `https_proxy` environment variable. It is needed both when installing pip itself and when using it to install Robot Framework and other Python packages.

How to set the `https_proxy` depends on the operating system similarly as configuring PATH. The value of this variable must be an URL of the proxy, for example, `http://10.0.0.42:8080`.

1.3.3 Installing with pip

The standard Python package manager is pip, but there are also other alternatives such as Buildout and easy_install. These instructions only cover using pip, but other package managers ought be able to install Robot Framework as well.

Latest Python, Jython, IronPython and PyPy versions contain pip bundled in. Which versions contain it and how to possibly activate it is discussed in sections below. See pip project pages if for the latest installation instructions if you need to install it.

> **Note**
>
> Robot Framework 3.1 and newer are distributed as wheels, but earlier versions are available only as source distributions in tar.gz format. It is possible to install both using pip, but installing wheels is a lot faster.

> **Note**
>
> Only Robot Framework 2.7 and newer can be installed using pip. If you need an older version, you must use other installation approaches.

Installing pip for Python

Starting from Python 2.7.9, the standard Windows installer by default installs and activates pip. Assuming you also have configured PATH and possibly set https_proxy, you can run `pip install robotframework` right after Python installation. With Python 3.4 and newer pip is officially part of the interpreter and should be automatically available.

Outside Windows and with older Python versions you need to install pip yourself. You may be able to do it using system package managers like Apt or Yum on Linux, but you can always use the manual installation instructions found from the pip project pages.

If you have multiple Python versions with pip installed, the version that is used when the `pip` command is executed depends on which pip is first in the PATH. An alternative is executing the `pip` module using the selected Python version directly:

```
python -m pip install robotframework
python3 -m pip install robotframework
```

Installing pip for Jython

Jython 2.7 contain pip bundled in, but it needs to be activated before using it by running the following command:

```
jython -m ensurepip
```

Jython installs its pip into *<JythonInstallation>/bin* directory. Does running `pip install robotframework` actually use it or possibly some other pip version depends on which pip is first in the PATH. An alternative is executing the `pip` module using Jython directly:

```
jython -m pip install robotframework
```

Installing pip for IronPython

IronPython 2.7.5 and newer contain pip bundled in. With IronPython 2.7.9 and newer pip also works out-of-the-box, but with earlier versions it needs to be activated with `ipy -m ensurepip` similarly as with Jython.

With IronPython 2.7.7 and earlier you need to use `-X:Frames` command line option when activating pip like `ipy -X:Frames -m ensurepip` and also when using it. Prior to IronPython 2.7.9 there were problems creating possible start-up scripts when installing modules. Using IronPython 2.7.9 is highly recommended.

IronPython installs pip into *<IronPythonInstallation>/Scripts* directory. Does running `pip install robotframework` actually use it or possibly some other pip version depends on which pip is first in the PATH. An alternative is executing the `pip` module using IronPython directly:

```
ipy -m pip install robotframework
```

Installing pip for PyPy

Also PyPy contains pip bundled in. It is not activated by default, but it can be activated similarly as with the other interpreters:

```
pypy -m ensurepip
pypy3 -m ensurepip
```

If you have multiple Python versions with pip installed, the version that is used when the `pip` command is executed depends on which pip is first in the PATH. An alternative is executing the `pip` module using PyPy directly:

```
pypy -m pip
pypy3 -m pip
```

Using pip

Once you have pip installed, and have set https_proxy if you are behind a proxy, using pip on the command line is very easy. The easiest way to use pip is by letting it find and download packages it installs from the Python Package Index (PyPI), but it can also install packages downloaded from the PyPI separately. The most common usages are shown below and pip documentation has more information and examples.

```
# Install the latest version (does not upgrade)
pip install robotframework

# Upgrade to the latest version
pip install --upgrade robotframework

# Install a specific version
pip install robotframework==2.9.2

# Install separately downloaded package (no network connection needed)
pip install robotframework-3.0.tar.gz

# Install latest (possibly unreleased) code directly from GitHub
pip install https://github.com/robotframework/robotframework/archive/master.zip

# Uninstall
pip uninstall robotframework
```

Notice that pip 1.4 and newer will only install stable releases by default. If you want to install an alpha, beta or release candidate, you need to either specify the version explicitly or use the `--pre` option:

```
# Install 3.0 beta 1
pip install robotframework==3.0b1

# Upgrade to the latest version even if it is a pre-release
pip install --pre --upgrade robotframework
```

Notice that on Windows pip, by default, does not recreate robot.bat and rebot.bat start-up scripts if the same Robot Framework version is installed multiple times using the same Python version. This mainly causes problems when using virtual environments, but is something to take into account also if doing custom installations using pip. A workaround if using the `--no-cache-dir` option like `pip install --no-cache-dir robotframework`. Alternatively it is possible to ignore the start-up scripts altogether and just use `python -m robot` and `python -m robot.rebot` commands instead.

1.3.4 Installing from source

This installation method can be used on any operating system with any of the supported interpreters. Installing *from source* can sound a bit scary, but the procedure is actually pretty straightforward.

Getting source code

You typically get the source code by downloading a *source distribution* from PyPI. Starting from Robot Framework 3.1 the source distribution is a zip package and with earlier versions it is in tar.gz format. Once you have downloaded the package, you need to extract it somewhere and, as a result, you get a directory named `robotframework-<version>`. The directory contains the source code and a `setup.py` script needed for installing it.

An alternative approach for getting the source code is cloning project's GitHub repository directly. By default you will get the latest code, but you can easily switch to different released versions or other tags.

Installation

Robot Framework is installed from source using Python's standard `setup.py` script. The script is in the directory containing the sources and you can run it from the command line using any of the supported interpreters:

```
python setup.py install
jython setup.py install
ipy setup.py install
pypy 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.

1.3.5 Standalone JAR distribution

Robot Framework is also distributed as a standalone Java archive that contains both Jython and Robot Framework and only requires Java a dependency. It is an easy way to get everything in one package that requires no installation, but has a downside that it does not work with the normal Python interpreter.

The package is named `robotframework-<version>.jar` and it is available on the Maven central. After downloading the package, you can execute tests with it like:

```
java -jar robotframework-3.0.jar mytests.robot
java -jar robotframework-3.0.jar --variable name:value mytests.robot
```

If you want to post-process outputs using Rebot or use other built-in supporting tools, you need to give the command name `rebot`, `libdoc`, `testdoc` or `tidy` as the first argument to the JAR file:

```
java -jar robotframework-3.0.jar rebot output.xml
java -jar robotframework-3.0.jar libdoc MyLibrary list
```

For more information about the different commands, execute the JAR without arguments.

In addition to the Python standard library and Robot Framework modules, the standalone JAR versions starting from 2.9.2 also contain the PyYAML dependency needed to handle yaml variable files.

1.3.6 Manual installation

If you do not want to use any automatic way of installing Robot Framework, you can always install it manually following these steps:

1. Get the source code. All the code is in a directory (a package in Python) called *robot*. If you have a source distribution or a version control checkout, you can find it from the *src* directory, but you can also get it from an earlier installation.
2. Copy the source code where you want to.
3. Decide how to run tests.

1.3.7 Verifying installation

After a successful installation, you should be able to execute the created runner scripts with `--version` option and get both Robot Framework and interpreter versions as a result:

```
$ robot --version
Robot Framework 3.0 (Python 2.7.10 on linux2)

$ rebot --version
Rebot 3.0 (Python 2.7.10 on linux2)
```

If running the runner scripts fails with a message saying that the command is not found or recognized, a good first step is double-checking the PATH configuration. If that does not help, it is a good idea to re-read relevant sections from these instructions before searching help from the Internet or as asking help on robotframework-users mailing list or elsewhere.

Where files are installed

When an automatic installer is used, Robot Framework source code is copied into a directory containing external Python modules. On UNIX-like operating systems where Python is pre-installed the location of this directory varies. If you have installed the interpreter yourself, it is normally *Lib/site-packages* under the interpreter installation directory, for example, *C:\Python27\Lib\site-packages*. The actual Robot Framework code is in a directory named *robot*.

Robot Framework runner scripts are created and copied into another platform-specific location. When using Python on UNIX-like systems, they normally go to */usr/bin* or */usr/local/bin*. On Windows and with Jython and IronPython, the scripts are typically either in *Scripts* or *bin* directory under the interpreter installation directory.

1.3.8 Uninstallation

The easiest way to uninstall Robot Framework is using pip:

```
pip uninstall robotframework
```

A nice feature in pip is that it can uninstall packages even if they are installed from the source. If you do not have pip available or have done a manual installation to a custom location, you need to find where files are installed and remove them manually.

If you have set PATH or configured the environment otherwise, you need to undo those changes separately.

1.3.9 Upgrading

If you are using pip, upgrading to a new version requires either specifying the version explicitly or using the `--upgrade` option. If upgrading to a preview release, `--pre` option is needed as well.

```
# Upgrade to the latest stable version. This is the most common method.
pip install --upgrade robotframework

# Upgrade to the latest version even if it would be a preview release.
pip install --upgrade --pre robotframework

# Upgrade to the specified version.
pip install robotframework==2.9.2
```

When using pip, it automatically uninstalls previous versions before installation. If you are installing from source, it should be safe to just install over an existing installation. If you encounter problems, uninstallation before installation may help.

When upgrading Robot Framework, there is always a change that the new version contains backwards incompatible changes affecting existing tests or test infrastructure. Such changes are very rare in minor versions like 2.8.7 or 2.9.2, but more common in major versions like 2.9 and 3.0. Backwards incompatible changes and deprecated features are explained in the release notes, and it is a good idea to study them especially when upgrading to a new major version.

1.3.10 Executing Robot Framework

Using `robot` and `rebot` scripts

Starting from Robot Framework 3.0, tests are executed using the `robot` script and results post-processed with the `rebot` script:

```
robot tests.robot
rebot output.xml
```

Both of these scripts are installed as part of the normal installation and can be executed directly from the command line if PATH is set correctly. They are implemented using Python except on Windows where they are batch files.

Older Robot Framework versions do not have the `robot` script and the `rebot` script is installed only with Python. Instead they have interpreter specific scripts `pybot`, `jybot` and `ipybot` for test execution and `jyrebot` and `ipyrebot` for post-processing outputs. These scripts still work, but they will be deprecated and removed in the future.

Executing installed `robot` module

An alternative way to run tests is executing the installed `robot` module or its sub module `robot.run` directly using Python's -m command line option. This is especially useful if Robot Framework is used with multiple Python versions:

```
python -m robot tests.robot
python3 -m robot.run tests.robot
jython -m robot tests.robot
/opt/jython/jython -m robot tests.robot
```

The support for `python -m robot` approach is a new feature in Robot Framework 3.0, but the older versions support `python -m robot.run`. The latter must also be used with Python 2.6.

Post-processing outputs using the same approach works too, but the module to execute is `robot.rebot`:

```
python -m robot.rebot output.xml
```

Executing installed `robot` directory

If you know where Robot Framework is installed, you can also execute the installed *robot* directory or the *run.py* file inside it directly:

```
python path/to/robot/ tests.robot
jython path/to/robot/run.py tests.robot
```

Running the directory is a new feature in Robot Framework 3.0, but the older versions support running the *robot/run.py* file.

Post-processing outputs using the *robot/rebot.py* file works the same way too:

```
python path/to/robot/rebot.py output.xml
```

Executing Robot Framework this way is especially handy if you have done a manual installation.

1.3.11 Using virtual environments

Python virtual environments 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. Virtual environments can be created using the virtualenv tool or, starting from Python 3.3, using the standard venv module.

Robot Framework in general works fine with virtual environments. The only problem is that when using pip on Windows, `robot.bat` and `rebot.bat` scripts are not recreated by default. This means that if Robot Framework is installed into multiple virtual environments, the `robot.bat` and `rebot.bat` scripts in the latter ones refer to the Python installation in the first virtual environment. A workaround is using the `--no-cache-dir` option when installing. Alternatively the start-up scripts can be ignored and `python -m robot` and `python -m robot.rebot` commands used instead.

1.4 Demonstrations

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

Quick Start Guide
: Introduces the most important features of Robot Framework and acts as an executable demo.

Robot Framework demo
: Simple example test cases. Demonstrates also creating custom test libraries.

Web testing demo
: Demonstrates how to create tests and higher level keywords. The system under test is a simple web page that is tested using SeleniumLibrary.

SwingLibrary demo
: Demonstrates using SwingLibrary for testing Java GUI applications.

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

2 Creating test data

2.1 Test data syntax

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.

2.1.1 Files and directories

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

- Test cases are created in test case files.
- A test case file automatically creates a test suite containing the test cases in that file.
- A directory containing test case files forms a higher-level test suite. Such a test suite directory 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 configuring the created test suite.

In addition to this, there are:

- Test libraries containing the lowest-level keywords.
- Resource files with variables and higher-level user keywords.
- 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

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, resource files and variable files. 2) Defining metadata for test suites and test cases.
Variables	Defining variables that can be used elsewhere in the test data.
Test Cases	Creating test cases from available keywords.
Tasks	Creating tasks using available keywords. Single file can only contain either test cases or tasks.
Keywords	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.

Possible data before the first section is ignored.

Note

Prior to Robot Framework 3.1, section names were space-insensitive, meaning that spaces could be removed (e.g. `TestCases`) or extra spaces added (e.g. `S e t t i n g s`). This is now deprecated and only the format in the table above, case-insensitively, is supported.

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

Robot Framework test data can be defined in few different formats:

1. The most common approach is using the plain text format and store files using the *.robot* extension. Alternatively it is possible to use the *.txt* extension.
2. The TSV format can be used as long as files are compatible with the plain text format.
3. Plain text test data can be embedded into reStructuredText files.
4. Earlier Robot Framework versions supported test data in HTML format.

Prior to Robot Framework 3.1, all aforementioned file formats were parsed automatically unless the `--extension` option was used to limit parsing. In Robot Framework 3.1 automatically parsing other than `*.robot` files was deprecated, and in the future other files are parsed only if that is explicitly configured using the `--extension` option. The support for the HTML format has bee deprecated in general it will be removed altogether in the future.

Plain text format

The plain text format is the base for all supported Robot Framework data formats. Test data is parsed line by line, but long logical lines can be split if needed. In a single line different data items like keywords and their arguments are separated from each others using a separator. The most commonly used separator is two or more spaces, but it is also possible to use a pipe character surrounded with spaces (`|`). Depending on the separator we can talk about the space separated format and the pipe separated format, but same file can actually contain lines with both separators.

Possible literal tab characters are converted to two spaces before parsing lines otherwise. This allows using a single tab as a separator instead of multiple spaces. Notice, however, that multiple consecutive tabs are still considered to be a single separator. If an actual tab character is needed in the data, it must be escaped like `\t`.

Plain text files containing non-ASCII characters must be saved using the UTF-8 encoding.

Space separated format

In the space separated format two or more spaces (or one or more tab characters) act as a separator between different data items. 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 if it makes sense.

```
*** Settings ***
Documentation     Example using the space separated plain text format.
Library           OperatingSystem

*** Variables ***
${MESSAGE}        Hello, world!

*** Test Cases ***
My Test
    [Documentation]    Example test
    Log    ${MESSAGE}
    My Keyword    /tmp

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

*** Keywords ***
My Keyword
    [Arguments]    ${path}
    Directory Should Exist    ${path}
```

Because space is used as separator, all empty items and items containing only spaces must be escaped with backslashes or with built-in `${EMPTY}` and `${SPACE}` variables, respectively.

Tip

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

Pipe separated format

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 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 plain text format.
| Library            | OperatingSystem

| *** Variables ***  |
| ${MESSAGE}         | Hello, world!

| *** Test Cases *** |                 |              |
| My Test            | [Documentation] | Example test |
|                    | Log             | ${MESSAGE}   |
|                    | My Keyword      | /tmp         |
| Another Test       | Should Be Equal | ${MESSAGE}   | Hello, world!

| *** Keywords ***   |                          |          |
| My Keyword         | [Arguments]              | ${path}  |
|                    | Directory Should Exist   | ${path}  |
```

There is no need to escape empty cells (other than the trailing empty cells) when using the pipe separated format. 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 -l *.txt \| wc -l |
|                    | Should Be Equal | ${file count}   | 42                       |
```

Editing

Plain text files can be easily edited using normal text editors and IDEs. Many of these tools also have plugins that support syntax highlighting Robot Framework test data and may also provide other features such as keyword completion. Robot Framework specific editors like RIDE naturally support the plain text format as well.

As already mentioned, plain text files containing non-ASCII characters must be saved using the UTF-8 encoding.

Recognized extensions

The recommended extension for test case files in the plain text format is *.robot*. Files using this extension are parsed automatically. Also the *.txt* extension can be used, but starting from Robot Framework 3.1 the `--extension` option must be used to explicitly tell that these files should be parsed.

When creating resource files, it is possible to use the special *.resource* extension in addition to the aforementioned *.robot* and *.txt* extensions. This way resource files and test cases files are easily separated from each others.

> **Note**
>
> The *.resource* extension is new in Robot Framework 3.1.

TSV format

Files in the tab-separated values (TSV) format are typically edited in spreadsheet programs and, because the syntax is so simple, they are easy to generate programmatically. They are also pretty easy to edit using normal text editors and they work well in version control, but the plain text format is even better suited for these purposes.

Using the TSV format

Setting	*Value*	*Value*	*Value*
Documentation	Example using the TSV format.		
Library	OperatingSystem		
Variable	*Value*	*Value*	*Value*
${MESSAGE}	Hello, world!		

Test Case	*Action*	*Argument*	*Argument*
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!
Keyword	*Action*	*Argument*	*Argument*
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

The TSV format and the space separated variant of the plain text format are nearly identical, but earlier Robot Framework versions had slightly different parser for these formats. The differences were:

- The TSV parser did not require escaping empty intermediate cells.
- The TSV parser removed possible quotes around cells that may be added by spreadsheet programs.

The TSV parser was deprecated in Robot Framework 3.1 and it will be removed in the future. It is still possible to use the TSV format, but files must be fully compatible with the plain text format. This basically requires escaping all empty cells and configuring spreadsheet program or other tool saving TSV files not to add surrounding quotes to cells.

Editing test data

You can create and edit TSV files in any spreadsheet program, such as Microsoft Excel. Select the tab-separated format when you save the file. It is also a good idea to turn all automatic corrections off and configure the tool to treat all values in the file as plain text. As explained above, TSV files should also be saved so that no quotes are added around the cells.

TSV files are relatively easy to edit with any text editor, especially if the editor supports visually separating tabs from spaces. The TSV format is also supported by RIDE.

Like plain text files, TSV files containing non-ASCII characters must be saved using the UTF-8 encoding.

Recognized extensions

Files in the TSV format are customarily saved using the *.tsv* extension, but starting from Robot Framework 3.1 the `--extension` option must be used to explicitly tell that these files should be parsed. Another possibility is saving also these files using the the *.robot* extension, but this requires the file to be fully compatible with the plain text syntax.

reStructuredText format

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.

When using reST files with Robot Framework, test data is defined using code blocks. Earlier Robot Framework versions also supported using tables and converting reST files to HTML, but this was deprecated in Robot Framework 3.1.

> **Note**
>
> Using reST files with Robot Framework requires the Python docutils module to be installed.

Using code blocks

reStructuredText documents can contain code examples in so called code blocks. When these documents are compiled into HTML or other formats, the code blocks are syntax highlighted using Pygments. In standard reST code blocks are started using the `code` directive, but Sphinx uses `code-block` or `sourcecode` instead. The name of the programming language in the code block is given as an argument to the directive. For example, following code blocks contain Python and Robot Framework examples, respectively:

```
.. code:: python

   def example_keyword():
       print('Hello, world!')
```

```
.. code:: robotframework

   *** Test Cases ***
   Example Test
       Example Keyword
```

When Robot Framework parses reStructuredText files, it first searches for possible `code`, `code-block` or `sourcecode` blocks containing Robot Framework test data. If such code blocks are found, data they contain is written into an in-memory file and executed. All data outside the code blocks is ignored.

The test data in the code blocks must be defined using the plain text format. As the example below illustrates, both space and pipe separated variants are supported:

```
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    /tmp

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

Also this text is outside code blocks and ignored. Above block used
the space separated plain text format and the block below uses the pipe
separated variant.

.. code:: robotframework

   | *** Keyword ***  |                         |          |
   | My Keyword       | [Arguments]             | ${path} |
   |                  | Directory Should Exist | ${path} |
```

Using tables

Earlier Robot Framework versions supported using reStructuredText also so that test data was defined in tables. These files were then internally converted to HTML format before parsing them. This functionality was deprecated in Robot Framework 3.1 and will be removed in the future along with the general support for the HTML format.

Editing

Test data in reStructuredText files can be edited with any text editor, and many editors also provide automatic syntax highlighting for it.

Robot Framework requires reST files containing non-ASCII characters to be saved using the UTF-8 encoding.

Recognized extensions

Robot Framework supports reStructuredText files using both *.rst* and *.rest* extension. Starting from Robot Framework 3.1 the `--extension` option must be used to explicitly tell that these files should be parsed.

Syntax errors in reST source files

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.

HTML format

Earlier Robot Framework versions supported test data in HTML format but this support has been deprecated in Robot Framework 3.1. All test data in HTML format should be converted to the plain text format or other supported formats. This is typically easiest by using the built-in Tidy tool.

2.1.4 Rules for parsing the data

Ignored data

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

- All data before the first test data section. If the data format allows data between sections, also that is ignored.
- Data in the Comments section.
- All empty rows.
- All empty cells at the end of rows, unless they are escaped.
- All single backslashes (`\`) when not used for 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 keywords *Log* or *Comment*.

Escaping

The escape character in Robot Framework test data is the backslash (`\`) and additionally 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.	`\${notvar}`
`\@`	At sign, never starts a list variable.	`\@{notvar}`
`\&`	Ampersand, never starts a dictionary variable.	`\&{notvar}`
`\%`	Percent sign, never starts an environment variable.	`\%{notvar}`
`\#`	Hash sign, never starts a comment.	`\# not comment`
`\=`	Equal sign, never part of named argument syntax.	`not\=named`
`\\|`	Pipe character, not a separator in the pipe separated format.	`ls -l *.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 and String 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 `${\n}` can be used if operating system dependent line terminator is needed (`\r\n` on Windows and `\n` elsewhere).

Note

Possible un-escaped whitespace character after the `\n` is ignored. This means that `two lines\nhere` and `two lines\n here` are equivalent. The motivation for this is to allow wrapping long lines containing newlines when using the HTML format, but the same logic is used also with other formats. An exception to this rule is that the whitespace character is not ignored inside the extended variable syntax.

Handling empty cells

If empty values are needed as arguments for keywords or otherwise, they often need to be escaped to prevent them from being ignored. Empty trailing cells must be escaped regardless of the test data format, and when using the space separated format all empty values must be escaped.

Empty cells can be escaped either with the backslash character or with built-in variable `${EMPTY}`. The latter is typically recommended as it is easier to understand. All these cases are illustrated by the following examples:

```
*** Test Cases ***
Using backslash
    Do Something    first arg    \
Using ${EMPTY}
    Do Something    first arg    ${EMPTY}
Non-trailing empty
    Do Something    ${EMPTY}     second arg    # Escaping needed in space separated format
```

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.
- Leading and trailing spaces are ignored when using the pipe separated format.

In these cases spaces need to be escaped. Similarly as when escaping empty cells, it is possible to do that either by using the backslash character or by using the 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.

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 when more than one space is needed.

Dividing test data to several rows

If there is more data than readily fits a row, it possible to use ellipsis (`...`) to continue the previous line. In test case and keyword tables, the ellipsis must be preceded by at least one empty cell. In settings and variable tables, it can be placed directly under the setting or variable name. In all tables, all empty cells before the ellipsis are ignored.

Also suite, test or keyword documentation and value of test suite metadata can be too long to fit into one row nicely. These values can be split into multiple rows as well, and they will be joined together with newlines.

All the syntax discussed above is illustrated in the following examples. In the first three tables test data has not been split, and the following three illustrate how fewer columns are needed after splitting the data to several rows.

```
*** Settings ***
Documentation      This is documentation for this test suite.\nThis kind of documentation can often be get quite long...
Default Tags       default tag 1    default tag 2    default tag 3    default tag 4    default tag 5

*** Variable ***
@{LIST}            this    list    is    quite    long    and    items in it could also be long

*** 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      This is documentation for this test suite.
...                This kind of documentation can often be get quite long...
Default Tags       default tag 1    default tag 2    default tag 3
...                default tag 4    default tag 5

*** Variable ***
@{LIST}            this     list     is      quite    long     and
...                items in it could also be long

*** 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

This section describes the overall test case syntax. Organizing test cases into test suites using test case files and 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 section.

2.2.1 Test case syntax

Basic syntax

Test cases are constructed in test case tables from the available keywords. Keywords can be imported from test libraries or resource files, or created in the keyword table of the test case file itself.

The first column in the test case table 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 table. It is an error to have something between the table headers and the first test.

The second column normally has keyword names. An exception to this rule is setting variables from 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 when 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 table

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.

[Tags]
: Used for tagging test cases.

[Setup], *[Teardown]*
: Specify test setup and teardown.

[Template]
: Specifies the template keyword to use. The test itself will contain only data to use as arguments to that keyword.

[Timeout]
: Used for setting a test case timeout. Timeouts are discussed in their own section.

Note

Setting names are case-insensitive, but the format used above is recommended. Prior to Robot Framework 3.1, settings were also space-insensitive meaning that extra spaces could be added (e.g. `[T a g s]`). This is now deprecated and only the format above, case-insensitively, is supported. Possible space between brackets and the name (e.g. `[ Tags ]`) is 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 table

The Setting table 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.

Test Setup, *Test Teardown*
: The default values for test setup and teardown.

Test Template
: The default template keyword to use.

Test Timeout
: The default value for test case timeout. Timeouts are discussed in their own section.

2.2.2 Using arguments

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 and library keywords 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 tool, but the same information is available on documentation generated by generic documentation tools such as `javadoc`.

Positional arguments

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 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, takes no arguments.

```
*** Test Cases ***
Example
    Create Directory    ${TEMPDIR}/stuff
    Copy File    ${CURDIR}/file.txt    ${TEMPDIR}/stuff
    No Operation
```

Default values

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`, but with keywords implemented using Java there may be multiple implementations of the same keyword with different arguments instead. 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

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 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

The named argument syntax makes using arguments with default values more flexible, and allows explicitly labeling what a certain argument value means. Technically named arguments work exactly like 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 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, 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 in both named argument names and values. If the value is a single 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 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. See 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 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.

Naming arguments is supported by user keywords and by most test libraries. The only exception are Java based libraries that use the static library API. Library documentation generated with Libdoc has a note does the library support named arguments or not.

Named arguments example

The following example demonstrates using the named arguments syntax with library keywords, user keywords, and when importing the Telnet 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

Robot Framework supports *free named arguments*, often also called *free keyword arguments* or *kwargs*, similarly as Python supports **kwargs. What this means is that a keyword can receive all arguments that use the 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. How keywords specify that they accept free named arguments depends on the keyword type. For example, Python based keywords simply use `**kwargs` and user keywords use `&{kwargs}`.

Free named arguments support variables similarly as named arguments. 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 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 (`*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.

```
*** 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) section under 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 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

Starting from Robot Framework 3.1, keywords can accept argument that must always be named using the 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 syntax supported by Python 3.

For most parts named-only arguments work the same way as named arguments. The main difference is that libraries implemented with Python 2 using the static library API do not support this syntax.

As an example of using the named-only arguments with user keywords, here is a variation of the *Run Program* in the above 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

A totally different approach to specify arguments is embedding them into keyword names. This syntax is supported by both test library keywords and user keywords.

2.2.3 Failures

When test case fails

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 is executed, and then execution continues from the next test case. It is also possible to use special continuable failures if stopping test execution is not desired.

Error messages

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 to keep reports easier to read, but full error messages are always visible in log files as messages of the failed keywords.

By default error messages are normal text, but they can contain HTML formatting. 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

The test case name comes directly from the Test Case table: 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 `${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.

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 in documentation and variables can be used to make the documentation dynamic.

If documentation is split into multiple columns, cells in one row are concatenated together with spaces. This is mainly be useful when using the HTML format and columns are narrow. If documentation is split into multiple rows, the created documentation lines themselves are concatenated using newlines. Newlines are not added if a line already ends with a newline or an escaping backslash.

```
*** 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.

2.2.5 Tagging test cases

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, logs and, of course, in the test data, so they provide metadata to test cases.
- Statistics about test cases (total, passed, failed are automatically collected based on tags).
- With tags, you can include or exclude test cases to be executed.

- With tags, you can specify which test cases are considered critical.

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 table
: 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 table
: 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 table
: 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 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

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, and `robot:no-dry-run`, that can be used to disable the dry run mode. More usages are likely to be added in the future.

2.2.6 Test setup and teardown

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 for that purpose. An alternative solution is executing multiple keywords using the BuiltIn 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 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 table. Individual test cases can also have their own setup or teardown. They are defined with the *[Setup]* or *[Teardown]* settings in the test case table 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 table
    Do Something

Overridden setup
    [Documentation]    Own setup, teardown from setting table
    [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. 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

Test templates convert normal keyword-driven test cases into data-driven 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

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 table, 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 table, 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 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 arguments with default values or varargs, as well as using named arguments and free named arguments, work with templates exactly like they work otherwise. Using variables in arguments is also supported normally.

Templates with embedded arguments

Templates support a variation of the 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 example in the next section.

Templates with for loops

If templates are used with 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 and for
    [Template]    Example keyword
    FOR    ${item}    IN    @{ITEMS}
        ${item}    2nd arg
    END
    FOR    ${index}    IN RANGE    42
        1st arg    ${index}
    END
```

2.2.8 Different test case styles

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

Workflow tests, such as the *Valid Login* test described earlier, 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

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, 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 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.

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, 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

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 (ATDD) or Specification by Example.

One way to write these requirements/tests is *Given-When-Then* style popularized by 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 pass actual data to keyword implementations. User keywords support this by allowing embedding arguments into keyword name.

2.3 Creating tasks

In addition to test automation, Robot Framework can be used for other automation purposes, including robotic process automation (RPA). It has been always been possible, bu 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 and the only real difference is in terminology. Tasks can also be organized into suites exactly like test cases.

2.3.1 Task syntax

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. The main difference is that tasks are created in task sections (or tables) 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

Settings that can be used in the task section are exactly the same as in the test case section. 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

Robot Framework 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, but the terminology differs.

2.4.1 Test case files

Robot Framework test cases are created using test case tables 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 is used, where one test case consists of only one high-level keyword.

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

Documentation
: Used for specifying a test suite documentation

Metadata
: Used for setting free test suite metadata as name-value pairs.

Suite Setup, *Suite Teardown*
: Specify suite setup and teardown.

> **Note**
>
> All setting names can optionally include a colon at the end, for example *Documentation:*. This can make reading the settings easier especially when using the plain text format.

Note

Setting names are case-insensitive, but the format used above is recommended. Prior to Robot Framework 3.1, settings were also space-insensitive meaning that spaces could be removed (e.g. `SuiteSetup`) or extra spaces added (e.g. `M e t a d a t a`). This is now deprecated and only the format above, case-insensitively, is supported.

2.4.2 Test suite directories

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 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) and the processing continues.

Initialization files

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 (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 tables 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 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, but setting some test case related settings 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, 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

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 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 table. 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.

```
*** 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.

2.4.4 Free test suite metadata

Test suites can also have other metadata than the documentation. This metadata is defined in the Setting table 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 (joined together with spaces) or into several rows (joined together with newlines), simple HTML formatting works and even 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.

2.4.5 Suite setup and teardown

Not only test cases 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.

Similarly as with test cases, a suite setup and teardown are keywords that may take arguments. They are defined in the Setting table 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

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. This section explains how to take test libraries into use and how to use the keywords they provide. Creating test libraries is described in a separate section.

2.5.1 Importing libraries

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

Using `Library` setting

Test libraries are normally imported using the *Library* setting in the Setting table 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. 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, resource files and test suite 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

Another possibility to take a test library into use is using the keyword *Import Library* from the BuiltIn 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

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 is the library imported using the *Library* setting or the *Import Library* keyword.

Using library name

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. 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, 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

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. 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. For Python libraries the extension is naturally *.py* and for Java libraries it can either be *.class* or *.java*, but the class file must always be available. If Python 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    /absolute/path/JavaLibrary.java
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. Additionally, importing libraries distributed in JAR or ZIP packages is not possible with this mechanism.

2.5.3 Setting custom name to test library

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. 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. This can happen, for example, if a Java library has a long package name.
- 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 table and when using the *Import Library* keyword.

2.5.4 Standard libraries

Some test libraries are distributed with Robot Framework and these libraries are called *standard libraries*. The BuiltIn 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

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

- BuiltIn
- Collections
- DateTime
- Dialogs
- OperatingSystem
- Process
- Screenshot
- String

- Telnet
- XML

Remote library

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 section for more information about this concept.

2.5.5 External libraries

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 and SwingLibrary, which are not packaged with the core framework. A list of publicly available libraries can be found from http://robotframework.org.

Generic and custom libraries can obviously also be implemented by teams using Robot Framework. See 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

2.6.1 Introduction

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 tables and keyword tables, but also all settings allow variables in their values. A normal keyword name *cannot* be specified with a variable, but the BuiltIn keyword *Run Keyword* can be used to get the same effect.

Robot Framework has its own variables that can be used as scalars, lists or dictionaries using syntax `${SCALAR}`, `@{LIST}` and `&{DICT}`, respectively. In addition to this, 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 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 as in `\${NAME}`.

2.6.2 Using variables

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.

Scalar variable syntax

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 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 Java object:

```
public class MyObj {

    public String toString() {
        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

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 individual list items are passed in as arguments separately. 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.

Prior to Robot Framework 2.9, scalar and list variables were stored separately, but it was possible to use list variables as scalars and scalar variables as lists. This caused lot of confusion when there accidentally was a scalar variable and a list variable with same name but different value.

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. 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

As discussed above, a variable containing a list can be used as a 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 individual items of the dictionary are passed as 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}
```

Dictionary variables are new in Robot Framework 2.9.

Using dictionary variables with other data

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

It is possible to access items of lists and dictionaries using special syntax `${var}[item]`. Accessing items is an old feature, but prior to Robot Framework 3.1 the syntax was `@{var}[item]` with lists and `&{var}[item]` with dictionaries. The old syntax still works in Robot Framework 3.1, but it will be deprecated in Robot Framework 3.2 and its meaning will change in Robot Framework 3.3.

Accessing list items

It is possible to access a certain item of a list variable 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. List items accessed in this manner can be used similarly as scalar variables.

```
*** Test Cases ***
List variable item
    Login    ${USER}[0]    ${USER}[1]
    Title Should Be    Welcome ${USER}[0]!

Negative index
    Log    ${LIST}[-1]

Index defined as variable
    Log    ${LIST}[${INDEX}]
```

List item access supports also the same "slice" functionality as Python with syntax like `${var}[1:]`. With this syntax you do not get a single item but a slice of the original list. Same way as with Python you can specify the start index, the end index, and the step:

```
*** Test Cases ***
Start index
    Keyword    ${LIST}[1:]

End index
    Keyword    ${LIST}[:4]

Start and end
    Keyword    ${LIST}[2:-1]

Step
    Keyword    ${LIST}[::2]
    Keyword    ${LIST}[2:-1:2]
```

> **Note**
>
> The slice syntax is new in Robot Framework 3.1 and does not work with the old `@{var}[index]` syntax.

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 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 list and dictionary structures 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

Robot Framework allows using environment variables in the test data using the syntax `%{ENV_VAR_NAME}`. They are limited to string values.

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 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
```

Java system properties

When running tests with Jython, it is possible to access Java system properties using same syntax as environment variables. If an environment variable and a system property with same name exist, the environment variable will be used.

```
*** Test Cases ***
System properties
    Log    %{user.name} running tests on %{os.name}
```

2.6.3 Creating variables

Variables can spring into existence from different sources.

Variable table

The most common source for variables are Variable tables in test case files and resource files. Variable tables 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 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 table 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 to multiple columns and rows. By default cells are catenated together using a space, but this can be changed by having `SEPARATOR=<sep>` in the first cell.

```
*** Variables ***
${EXAMPLE}       This value is joined    together with a space
${MULTILINE}     SEPARATOR=\n    First line
...              Second line     Third line
```

Joining long values like above is a new feature in Robot Framework 2.9.

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 table 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.

```
*** 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 table 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 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 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.

> **Note**
>
> Starting from Robot Framework 3.0.3, dictionary variable keys are accessible recursively 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 with 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

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.

Setting variables in command line

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 table 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 `${USER}` get the values `localhost:7272` and `robot`
- `${ESCAPED}` gets the value `"quotes and spaces"`

The basic syntax for taking variable files into use from the command line is `--variablefile path/to/variables.py`, and 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.

Return values from keywords

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 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 keywords as explained in the next section.

Assigning scalar variables

Any value returned by a keyword can be assigned to a 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 if it has a list-like value and as a 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:

```
*** 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:

```
*** 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 in the variable table. 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.

The support for assigning multiple variables was slightly changed in Robot Framework 2.9. Prior to it a list variable was only allowed as the last assigned variable, but nowadays it can be used anywhere. Additionally, it was possible to return more values than scalar variables. In that case the last scalar variable was magically turned into a list containing the extra values.

Using *Set Test/Suite/Global Variable* keywords

The BuiltIn 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.

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 table in the test data file or importing them from 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 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 and return nothing. On the other hand, another BuiltIn keyword *Set Variable* sets local variables using return values.

2.6.4 Built-in variables

Robot Framework provides some built-in variables that are available automatically.

Operating-system variables

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

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

Also Boolean values and Python `None` and Java `null` 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

Null
    ${ret} =    Get Value    arg        # Checking that Get Value returns Java null
    Should Be Equal    ${ret}    ${null}
```

These variables are case-insensitive, so for example `${True}` and `${true}` are equivalent. Additionally, `${None}` and `${null}` are synonyms, because when running tests on the Jython interpreter, Jython automatically converts `None` and `null` to the correct format when necessary.

Space and empty variables

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 with a backslash. If more than one space is needed, it is possible to use the 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 `@{EMPTY}` and an empty 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 when the template keyword is used without arguments 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 should be specified using the escape sequences like `\xA0` (NO-BREAK SPACE) and `\u3000` (IDEOGRAPHIC SPACE).

Note

`&{EMPTY}` is new in Robot Framework 2.9.

Automatic variables

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 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
${TEST MESSAGE}	The message of the current test case.	Test 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
${SUITE MESSAGE}	The full message of the current test suite, including statistics.	Suite teardown
${KEYWORD STATUS}	The status of the current keyword, either PASS or FAIL.	User keyword teardown
${KEYWORD MESSAGE}	The possible error message of the current keyword.	User

		keyword teardown
${LOG LEVEL}	Current log level.	Everywhere
${OUTPUT FILE}	An absolute path to the output file.	Everywhere
${LOG FILE}	An absolute path to the log file or string NONE when no log file is created.	Everywhere
${REPORT FILE}	An absolute path to the report file or string NONE when no report is created.	Everywhere
${DEBUG FILE}	An absolute path to the debug file or string NONE when no debug file is created.	Everywhere
${OUTPUT DIR}	An absolute path to the output directory.	Everywhere

Suite related variables `${SUITE SOURCE}`, `${SUITE NAME}`, `${SUITE DOCUMENTATION}` and `&{SUITE METADATA}` are available already when test 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

Variables coming from different sources have different priorities and are available in different scopes.

Variable priorities

Variables from the command line

> Variables set in the 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 tables 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 (`--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 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 table in a test case file

> Variables created using the Variable table 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 tables are available in all other tables in the file where they are created. This means that they can be used also in the Setting table, for example, for importing more variables from resource and variable files.

Imported resource and variable files

> Variables imported from the 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 table 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 table of the file that imports them. This is due to the Variable table being processed before the Setting table 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 or 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 like `${TEMPDIR}` and `${TEST_NAME}` have the highest priority of all variables. They cannot be overridden using Variable table or from command line, but even they can be reset during the test execution. An exception to this rule are 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

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 with the `--variable` and `--variablefile` options, but it is also possible to create new global variables or change the existing ones with the BuiltIn keyword *Set Global Variable* anywhere in the test data. Additionally also 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 tables, imported from resource and variable files, or set during the test execution using the BuiltIn 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 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 keyword *Set Test Variable* anywhere in a test case.

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 from executed keywords and user keywords also get them as arguments.

It is recommended to use lower-case letters with local variables.

> **Note**
>
> Prior to Robot Framework 2.9 variables in the local scope leaked to lower level user keywords. This was never an intended feature, and variables should be set or passed explicitly also with earlier versions.

2.6.6 Advanced variable features

Extended variable syntax

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 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.

If the object that is used is implemented with Java, the extended variable syntax allows you to access attributes using so-called bean properties. In essence, this means that if you have an object with the `getName` method set into a variable `${OBJ}`, then the syntax `${OBJ.name}` is equivalent to but clearer than `${OBJ.getName()}`. The Python object used in the previous example could thus be replaced with the following Java implementation:

```
public class MyObject:

    private String name;

    public MyObject(String name) {
        name = name;
    }

    public String getName() {
        return name;
    }

    public String eat(String what) {
        return name + " eats " + what;
    }

    public String toString() {
        return name;
    }
}
```

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 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

It is possible to set attributes of objects stored to scalar variables using keyword return values and a variation of the 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, 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

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}
```

2.7 Creating user keywords

Keyword tables 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

Basic syntax

In many ways, the overall user keyword syntax is identical to the test case syntax. User keywords are created in keyword tables which differ from test case tables 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, similarly as user keyword return values.

User keywords can be created in test case files, resource files, and test suite 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 table

User keywords can have similar settings as test cases, 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.

[Tags]
: Sets tags for the keyword.

[Arguments]
: Specifies user keyword arguments.

[Return]
: Specifies user keyword return values.

[Teardown]
: Specify user keyword teardown.

[Timeout]
: Sets the possible user keyword timeout. Timeouts are discussed in a section of their own.

> **Note**
>
> Setting names are case-insensitive, but the format used above is recommended. Prior to Robot Framework 3.1, settings were also space-insensitive meaning that extra spaces could be added (e.g. `[T a g s]`). This is now deprecated and only the format above, case-insensitively, is supported. Possible space between brackets and the name (e.g. `[ Tags ]`) is still allowed.

2.7.2 User keyword name and documentation

The user keyword name is defined in the first column of the user keyword table. 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. This setting documents the user keyword in the test data. It is also shown in a more formal keyword documentation, which the Libdoc tool can create from resource files. Finally, the first logical row of the documentation, until the first empty row, is shown as a keyword documentation in test logs.

```
*** 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 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

Starting from Robot Framework 2.9, keywords can also have tags. User keywords tags can be set with *[Tags]* setting similarly as 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, where the keywords can also be searched based on tags. The --removekeywords and --flattenkeywords 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, user keyword tags with `robot-` and `robot:` prefixes are reserved 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

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, for example `${arg}`.

Positional arguments with user keywords

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

When creating user keywords, positional arguments are sufficient in most situations. It is, however, sometimes useful that keywords have 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 created on test, suite or global scope, but local variables of the keyword executor cannot be used. Starting from Robot Framework 3.0, 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 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

Sometimes even default values are not enough and there is a need for a keyword accepting variable number of arguments. User keywords support also this feature. All that is needed is having 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. 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

User keywords can also accept free named arguments by having a dictionary variable like `&{named}` as the absolutely last argument. When the keyword is called, this variable will get all named arguments that do not match any positional argument or named-only argument 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 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

Starting from Robot Framework 3.1, user keywords support named-only arguments that are inspired by Python 3 keyword-only arguments. This syntax is typically used by having normal arguments *after* variable number of arguments (`@{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 as well as with free named arguments. 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. 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

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

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

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 too 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 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 . 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

When keywords with embedded arguments are called, the values are matched internally using regular expressions (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. 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 ${a:\d+} ${operator:[+-]} ${b:\d+}
    Calculate    ${a}    ${operator}    ${b}

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. 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.

Escaping special characters

There are some special characters that need to be escaped when used in the custom embedded arguments regexp. First of all, possible closing curly braces (`}`) in the pattern need to be escaped with a single backslash (`\}`) because otherwise the argument would end already there. This is illustrated in the previous example with keyword *Today is ${date:\d{4\}-\d{2\}-\d{2\}}*.

Backslash (`\`) is a special character in Python regular expression syntax and thus needs to be escaped

if you want to have a literal backslash character. The safest escape sequence in this case is four backslashes (`\\\\`) but, depending on the next character, also two backslashes may be enough.

Notice also that keyword names and possible embedded arguments in them should *not* be escaped using the normal test data escaping rules. This means that, for example, backslashes in expressions like `${name:\w+}` should not be escaped.

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

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. The example below illustrates this. Notice also that prefixes *Given*, *When* and *Then* are left out of the keyword definitions.

```
*** 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.

2.7.6 User keyword return values

Similarly as library keywords, also user keywords can return values. Typically return values are defined with the *[Return]* setting, but it is also possible to use BuiltIn keywords *Return From Keyword* and *Return From Keyword If*. Regardless how values are returned, they can be assigned to variables in test cases and in other user keywords.

Using *[Return]* setting

The most common case is that a user keyword returns one value and it is assigned to a scalar variable. When using the *[Return]* setting, this is done by having the return value in the next cell after the setting.

User keywords can also return several values, which can then be assigned into several scalar variables at once, to a list variable, or to scalar variables and a list variable. Several values can be returned simply by specifying those values in different cells after the *[Return]* setting.

```
*** Test Cases ***
One Return Value
    ${ret} =    Return One Value    argument
    Some Keyword    ${ret}

Multiple Values
    ${a}    ${b}    ${c} =    Return Three Values
    @{list} =    Return Three Values
    ${scalar}    @{rest} =    Return Three Values

*** Keywords ***
Return One Value
    [Arguments]    ${arg}
    Do Something    ${arg}
    ${value} =    Get Some Value
    [Return]    ${value}

Return Three Values
    [Return]    foo    bar    zap
```

Using special keywords to return

BuiltIn 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]* setting discussed above.

The first example below is functionally identical to the previous *[Return]* setting example. The second, and more advanced, example demonstrates returning conditionally inside a for loop.

```
*** Test Cases ***
One Return Value
    ${ret} =    Return One Value  argument
    Some Keyword    ${ret}

Advanced
    @{list} =    Create List    foo    baz
    ${index} =    Find Index    baz    @{list}
    Should Be Equal    ${index}    ${1}
    ${index} =    Find Index    non existing    @{list}
    Should Be Equal    ${index}    ${-1}

*** Keywords ***
Return One Value
    [Arguments]    ${arg}
    Do Something    ${arg}
    ${value} =    Get Some Value
    Return From Keyword    ${value}
    Fail    This is not executed

Find Index
    [Arguments]    ${element}    @{items}
    ${index} =    Set Variable    ${0}
    FOR    ${item}    IN    @{items}
        Return From Keyword If    '${item}' == '${element}'    ${index}
        ${index} =    Set Variable    ${index + 1}
    END
    Return From Keyword    ${-1}    # Could also use [Return]
```

2.7.7 User keyword teardown

User keywords may have a teardown defined using *[Teardown]* setting.

Keyword teardown works much in the same way as a test case 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

User keywords and variables in test case files and test suite initialization files can only be used in files where they are created, but *resource files* provide a mechanism for sharing them. Since the resource file structure is very close to test case files, it is easy to create them.

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 code, which also makes them somewhat more complicated than Variable tables.

2.8.1 Resource files

Taking resource files into use

Resource files are imported using the *Resource* setting in the Settings table. The path to the resource file is given in the cell after the setting name.

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. The path can contain variables, and it is recommended to use them to make paths system-independent (for example, *${RESOURCES}/login_resources.robot* or *${RESOURCE_PATH}*). Additionally, forward slashes (`/`) in the path are automatically changed to backslashes (`\`) on Windows.

Resource files can use all the same extensions as test case files created using the supported file formats. When using the plain text format, it is possible to use a special *.resource* extension in addition to the normal *.robot* extensions. This makes it easier to separate test case files and resource files from each others.

```
*** 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

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 tables. Additionally, the Setting table in resource files can contain only import settings (*Library*, *Resource*, *Variables*) and *Documentation*. The Variable table and Keyword table 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 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

Keywords created in a resource file can be documented using *[Documentation]* setting. The resource file itself can have *Documentation* in the Setting table similarly as test suites.

Both Libdoc and 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

```
*** 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

Variable files contain variables that can be used in the test data. Variables can also be created using variable tables or set from the command line, but variable files allow creating them dynamically and their variables can contain any objects.

Variable files are typically implemented as Python modules and there are two different approaches for creating variables:

Creating variables directly
: 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 the value.

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 Python or Java classes that the framework will instantiate. Also in this case it is possible to create variables as attributes or get them from a special method.

Taking variable files into use

Setting table

All test data files can import variables using the *Variables* setting in the Setting table, in the same way as resource files are imported 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 the directories in the module search path. The path can also contain variables, and slashes are converted to backslashes on Windows. If an argument file takes arguments, they are specified in the cells after the path and also they can contain variables.

```
*** Settings ***
Variables    myvariables.py
Variables    ../data/variables.py
Variables    ${RESOURCES}/common.py
Variables    taking_arguments.py    arg1    ${ARG2}
```

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 tables 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 to them, and possible arguments are joined to the path with a colon (`:`):

```
--variablefile myvariables.py
--variablefile path/variables.py
--variablefile /absolute/path/common.py
--variablefile taking_arguments.py:arg1:arg2
```

Variable files taken into use from the command line are also searched from the module search path similarly as variable files imported in the Setting table.

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 "myvariables.py;argument:with:colons"
--variablefile C:\path\variables.py;D:\data.xls
```

Variables in these variable files are globally available in all test data files, similarly as individual variables 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.

Creating variables directly

Basic syntax

When variable files are taken into use, they are imported as Python modules and all their global attributes that do not start with an underscore (`_`) are 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, lists and the dictionary also a list variable like `@{STRINGS}` (in the dictionary's case that variable would only contain keys), and the dictionary also as a 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 in the Variable table. 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 table 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 tables. Instead, their variables can contain any objects. In the example below, the variable `${MAPPING}` contains a Java Hashtable with two values (this example works only when running tests on Jython).

```
from java.util import Hashtable

MAPPING = Hashtable()
MAPPING.put("one", 1)
MAPPING.put("two", 2)
```

The second example creates `${MAPPING}` as 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 *`.

Getting variables from a special function

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 or a Java `Map` with variable names as keys and variable values as values. Created variables can be used as scalars, lists, and dictionaries exactly like when creating variables directly, 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 creating variables directly example.

```
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 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 Python or Java class

It is possible to implement variables files also as Python or Java classes.

Implementation

Because variable files are always imported using a file system path, creating them as classes has some restrictions:

- Python classes must have the same name as the module they are located.
- Java classes must live in the default package.
- Paths to Java classes must end with either *.java* or *.class*. The class file must exists in both cases.

Regardless the implementation language, 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` (or `getVariables`) 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 using both Python and Java. Both of them create variables `${VARIABLE}` and `@{LIST}` from class attributes and `${ANOTHER VARIABLE}` from an instance attribute.

```
class StaticPythonExample(object):
    variable = 'value'
    LIST__list = [1, 2, 3]
    _not_variable = 'starts with an underscore'

    def __init__(self):
```

```
        self.another_variable = 'another value'
```

```
public class StaticJavaExample {
    public static String variable = "value";
    public static String[] LIST__list = {1, 2, 3};
    private String notVariable = "is private";
    public String anotherVariable;

    public StaticJavaExample() {
        anotherVariable = "another value";
    }
}
```

The second examples utilizes dynamic approach for getting variables. Both of them create only one variable `${DYNAMIC VARIABLE}`.

```
class DynamicPythonExample(object):

    def get_variables(self, *args):
        return {'dynamic variable': ' '.join(args)}
```

```
import java.util.Map;
import java.util.HashMap;

public class DynamicJavaExample {

    public Map<String, String> getVariables(String arg1, String arg2) {
        HashMap<String, String> variables = new HashMap<String, String>();
        variables.put("dynamic variable", arg1 + " " + arg2);
        return variables;
    }
}
```

Variable file as YAML

Variable files can also be implemented as YAML 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 module to be installed. If you have pip installed, you can install it simply by running `pip install pyyaml`.
>
> YAML support is new in Robot Framework 2.9. Starting from version 2.9.2, the standalone JAR distribution has PyYAML included by default.

YAML variable files can be used exactly like normal variable files from the command line using `--variablefile` option, in the settings table using *Variables* setting, and dynamically using the *Import Variables* keyword. The only thing to remember is that paths to YAML files must always end with *.yaml* extension.

If the above YAML file is imported, it will create exactly the same variables as the following variable table:

```
*** 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 in the variable table. 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 Advanced features

2.9.1 Handling keywords with same names

Keywords that are used with Robot Framework are either library keywords or user keywords. The former come from standard libraries or external libraries, and the latter are either created in the same file where they are used or then imported from 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

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 same file where it is used. 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

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 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, 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

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 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.

2.9.2 Timeouts

Keywords may be problematic in situations where they take exceptionally long to execute or just hang endlessly. Robot Framework allows you to set timeouts both for test cases and 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 or 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, if necessary.

Test case timeout

The test case timeout can be set either by using the *Test Timeout* setting in the Setting table or the *[Timeout]* setting in the Test Case table. *Test Timeout* in the Setting table defines a default test timeout value for all the test cases in the test suite, whereas *[Timeout]* in the Test Case table applies a timeout to an individual 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 value `NONE` for this purpose.

Regardless of where the test timeout is defined, the first cell after the setting name contains the duration of the timeout. The duration must be given in Robot Framework's time format, that is, either directly in seconds or in a format like `1 minute 30 seconds`. It must be noted that there is always some overhead by the framework, and timeouts shorter than one second are thus not recommended.

The default error message displayed when a test timeout occurs is `Test timeout <time> exceeded`. It is also possible to use custom error messages, and these messages are written into the cells after the timeout duration. The message can be split into multiple cells, similarly as documentations. Both the timeout value and the error message may contain variables.

If there is a timeout, the keyword running is stopped at the expiration of the timeout and the test case fails. However, keywords executed as test teardown are not interrupted if a test timeout occurs, because they are normally engaged in important clean-up activities. If necessary, it is possible to interrupt also these keywords with user keyword timeouts.

```
*** Settings ***
Test Timeout    2 minutes

*** Test Cases ***
Default Timeout
    [Documentation]    Timeout from the Setting table is used
    Some Keyword    argument

Override
    [Documentation]    Override default, use 10 seconds timeout
    [Timeout]    10
    Some Keyword    argument

Custom Message
    [Documentation]    Override default and use custom message
    [Timeout]    1min 10s    This is my custom error
    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

A timeout can be set for a user keyword using the *[Timeout]* setting in the Keyword table. The syntax for setting it, including how timeout values and possible custom messages are given, is identical to the syntax used with test case timeouts. If no custom message is provided, the default error message `Keyword timeout <time> exceeded` is used if a timeout occurs.

Starting from Robot Framework 3.0, timeout can be specified as a variable so that the variable value is given as an argument. Using global variables works already with previous versions.

```
*** Keywords ***
Timed Keyword
    [Documentation]    Set only the timeout value and not the custom message.
    [Timeout]    1 minute 42 seconds
    Do Something
    Do Something Else

Wrapper With Timeout
    [Arguments]    @{args}
    [Documentation]    This keyword is a wrapper that adds a timeout to another keyword.
    [Timeout]    2 minutes    Original Keyword didn't finish in 2 minutes
    Original Keyword    @{args}

Wrapper With Customizable Timeout
    [Arguments]    ${timeout}    @{args}
    [Documentation]    Same as the above but timeout given as an argument.
    [Timeout]    ${timeout}
    Original Keyword    @{args}
```

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.

2.9.3 For loops

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

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, including 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 and the loop is run altogether ten times.

It is often convenient to use for loops with 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

For loop syntax was enhanced in various ways in Robot Framework 3.1. The most noticeable change was that loops nowadays end with the explicit `END` marker and keywords inside the loop do not need to be indented. In the space separated plain text format indentation required escaping with a backslash which resulted in quite ugly syntax:

```
*** Test Cases ***
Example
    :FOR    ${animal}    IN    cat    dog
    \    Log    ${animal}
    \    Log    2nd keyword
    Log    Outside loop
```

Another change, also visible in the example above, was that the for loop marker used to be `:FOR` when nowadays just `FOR` is enough. Related to that, the `:FOR` marker and also the `IN` separator were case-insensitive but nowadays both `FOR` and `IN` are case-sensitive.

Old for loop syntax still works in Robot Framework 3.1 and only using `IN` case-insensitively causes a deprecation warning. Not closing loops with `END`, escaping keywords inside loops with `\`, and using `:FOR` instead of `FOR` are all going to be deprecated in Robot Framework 3.2. Users are advised to switch to the new syntax as soon as possible.

Nested for loops

Having nested for loops is not supported directly, but it is possible to use a user keyword inside a for loop and have another for loop there.

```
*** Keywords ***
Handle Table
    [Arguments]    @{table}
    FOR    ${row}    IN    @{table}
        Handle Row    @{row}
    END

Handle Row
    [Arguments]    @{row}
    FOR    ${cell}    IN    @{row}
        Handle Cell    ${cell}
    END
```

Using several loop variables

It is also possible to use several loop variables. The syntax is the same as with the normal for loop, but all loop variables are listed in the cells between `FOR` and `IN`. There can be any number of loop variables, but the number of values must be evenly dividable by the number of 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 ***
Three loop variables
    FOR    ${index}    ${english}    ${finnish}    IN
    ...    1           cat           kissa
    ...    2           dog           koira
    ...    3           horse         hevonen
        Add to dictionary    ${english}    ${finnish}    ${index}
    END
    FOR    ${name}    ${id}    IN    @{EMPLOYERS}
        Create    ${name}    ${id}
    END
```

For-in-range loop

Earlier for loops always iterated over a sequence, and this is also the most common use case. Sometimes it is still convenient to have a for loop that is executed a certain number of times, and Robot Framework has a special `FOR index IN RANGE limit` syntax for this purpose. This syntax is derived from the similar Python idiom using the built-in range() function.

Similarly as other for loops, the for-in-range loop starts with `FOR` and the loop variable is in the next cell. In this format there can be only one loop variable and it contains the current loop index. The next cell must contain `IN RANGE` (case-sensitive) and the subsequent cells loop limits.

In the simplest case, only the upper limit of the loop is specified. In this case, loop indexes 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 indexes 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
```

Note

Prior to Robot Framework 3.1, the `IN RANGE` separator was both case- and space-insensitive. Such usage is nowadays deprecated and exactly `IN RANGE` is required.

For-in-enumerate loop

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.

For-in-enumerate loops work just like regular for loops, except the cell after its loop variables must say `IN ENUMERATE` (case-sensitive), and they must have an additional index variable before any other loop-variables. That index variable has a value of `0` for the first iteration, `1` for the second, etc.

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
```

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 first, 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
```

Note

Prior to Robot Framework 3.1, the `IN ENUMERATE` separator was both case- and space-insensitive. Such usage is nowadays deprecated and exactly `IN ENUMERATE` is required.

For-in-zip loop

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.

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    ${idx}    IN RANGE    ${length}
        Number Should Be Named    ${NUMBERS}[${idx}]    ${NAMES}[${idx}]
    END

For-in-zip
    FOR    ${number}    ${name}    IN ZIP    ${NUMBERS}    ${NAMES}
        Number Should Be Named    ${number}    ${name}
    END
```

Similarly as for-in-range and for-in-enumerate loops, for-in-zip loops require the cell after the loop variables to read `IN ZIP` (case-sensitive).

Values used with for-in-zip loops must be lists or list-like objects, and there must be same number of loop variables as lists to loop over. Looping will stop when the shortest list is exhausted.

Note that any lists used with for-in-zip should usually be given as scalar variables like `${list}`. A list variable only works if its items themselves are lists.

Note

Prior to Robot Framework 3.1, the `IN ZIP` separator was both case- and space-insensitive. Such usage is nowadays deprecated and exactly `IN ZIP` is required.

Exiting for loop

Normally for loops are executed until all the loop values have been iterated or a keyword used inside the loop fails. If there is a need to exit the loop earlier, BuiltIn keywords *Exit For Loop* and *Exit For Loop If* can be used to accomplish that. They works similarly as `break` statement in Python, Java, and many other programming languages.

Exit For Loop and *Exit For Loop If* keywords can be used directly inside a for loop or in a keyword that the loop uses. In both cases test execution continues after the loop. It is an error to use these keywords outside a for loop.

```
*** Test Cases ***
Exit Example
    ${text} =    Set Variable    ${EMPTY}
    FOR    ${var}    IN    one    two
        Run Keyword If    '${var}' == 'two'    Exit For Loop
        ${text} =    Set Variable    ${text}${var}
    END
    Should Be Equal    ${text}    one
```

In the above example it would be possible to use *Exit For Loop If* instead of using *Exit For Loop* with *Run Keyword If*. For more information about these keywords, including more usage examples, see their documentation in the BuiltIn library.

Continuing for loop

In addition to exiting a for loop prematurely, it is also possible to continue to the next iteration of the loop before all keywords have been executed. This can be done using BuiltIn keywords *Continue For Loop* and *Continue For Loop If*, that work like `continue` statement in many programming languages.

Continue For Loop and *Continue For Loop If* keywords can be used directly inside a for loop or in a

keyword that the loop uses. In both cases rest of the keywords in that iteration are skipped and execution continues from the next iteration. If these keywords are used on the last iteration, execution continues after the loop. It is an error to use these keywords outside a for loop.

```
*** Test Cases ***
Continue Example
    ${text} =    Set Variable    ${EMPTY}
    FOR    ${var}    IN    one    two    three
        Continue For Loop If    '${var}' == 'two'
        ${text} =    Set Variable    ${text}${var}
    END
    Should Be Equal    ${text}    onethree
```

For more information about these keywords, including usage examples, see their documentation in the BuiltIn library.

Removing unnecessary keywords from outputs

For loops with multiple iterations often create lots of output and considerably increase the size of the generated output and log files. It is possible to remove unnecessary keywords from the outputs using `--RemoveKeywords FOR` command line option.

Repeating single keyword

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 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
```

2.9.4 Conditional execution

In general, it is not recommended to have conditional logic in test cases, or even in user keywords, because it can make them hard to understand and maintain. Instead, this kind of logic should be in test libraries, where it can be implemented using natural programming language constructs. However, some conditional logic can be useful at times, and even though Robot Framework does not have an actual if/else construct, there are several ways to get the same effect.

- The name of the keyword used as a setup or a teardown of both test cases and test suites can be specified using a variable. This facilitates changing them, for example, from the command line.
- The BuiltIn 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 keywords *Run Keyword If* and *Run Keyword Unless* execute a named keyword only if a certain expression is true or false, respectively. They are ideally suited to creating simple if/else constructs. For an example, see the documentation of the former.
- Another BuiltIn keyword, *Set Variable If*, can be used to set variables dynamically based on a given expression.
- There are several BuiltIn keywords that allow executing a named keyword only if a test case or test suite has failed or passed.

2.9.5 Parallel execution of keywords

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 OperatingSystem library keywords *Start Process* and *Read Process Output* for an example.

3 Executing test cases

3.1 Basic usage

Robot Framework test cases are executed from the command line, and the end result is, by default, an output file in XML format and an HTML report and log. After the execution, output files can be combined and otherwise post-processed with the Rebot tool.

3.1.1 Starting test execution

Synopsis

```
robot [options] data_sources
python|jython|ipy|pypy -m robot [options] data_sources
python|jython|ipy|pypy path/to/robot/ [options] data_sources
java -jar robotframework.jar [options] data_sources
```

Test execution is normally started using the `robot` runner script. Alternatively it is possible to execute the installed robot module or robot directory directly using the selected interpreter. The final alternative is using the standalone JAR distribution.

> **Note**
>
> The `robot` script is new in Robot Framework 3.0. Prior to that, there were `pybot`, `jybot` and `ipybot` scripts that executed tests using Python, Jython and IronPython, respectively. These scripts were removed in Robot Framework 3.1 and nowadays `robot` must be used regardless the interpreter.

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

Robot Framework test cases are created in files and 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, from the file or directory name. 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
```

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

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

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

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, 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

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

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.

> **Note**
>
> Support for adding or dropping `no` prefix is a new feature in Robot Framework 2.9. In earlier versions options accepting no values could be disabled by using the exact same option again.

Simple patterns

Many command line options take arguments as *simple patterns*. These glob-like patterns 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`.

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

Tag patterns

Most tag related options accept arguments as *tag patterns*. They have all the same characteristics as simple patterns, 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'.
```

  Starting from Robot Framework 2.9 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

Environment variables `ROBOT_OPTIONS` and `REBOT_OPTIONS` can be used to specify default options for test execution and result post-processing, 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="--critical regression --tagdoc 'mytag:Example doc with spaces'"
robot tests.robot
export REBOT_OPTIONS="--reportbackground green:yellow:red"
rebot --name example output.xml
```

> **Note**
>
> Possibility to have spaces in values by surrounding them in quotes is new in Robot Framework 2.9.2.

3.1.3 Test results

Command line output

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 critical tests, 1 passed, 1 failed
2 tests total, 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

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.

Return codes

Runner scripts communicate the overall test execution status to the system running them using return codes. When the execution starts successfully and no critical test fail, the return code is zero. All possible return codes are explained in the table below.

Possible return codes

RC	Explanation
0	All critical tests passed.
1-249	Returned number of critical tests failed.
250	250 or more critical 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 critical 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.

Errors and warnings during execution

During the test execution there can be unexpected problems like failing to import a library or a resource file or a keyword being deprecated. 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. Normally these errors and warnings are generated by Robot Framework itself, but libraries can also log 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

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, `--argumentfile` cannot be given in shortened format like `--argumentf`. Additionally, using it case-insensitively like `--ArgumentFile` is only supported by Robot Framework 3.0.2 and newer.

Argument file syntax

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

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

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

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 or Jython version and the platform type:

```
$ robot --version
Robot Framework 3.1 (Jython 2.7.0 on java1.7.0_45)

C:\>rebot --version
Rebot 3.1 (Python 3.7.0 on win32)
```

3.1.6 Creating start-up scripts

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.

In the first examples, the same web tests are executed with different browsers and the results combined afterwards. This is easy with shell scripts, as practically you just list the needed commands one after another:

```
#!/bin/bash
robot --variable BROWSER:Firefox --name Firefox --log none --report none --output out/fx.xml login
robot --variable BROWSER:IE --name IE --log none --report none --output out/ie.xml login
rebot --name Login --outputdir out --output login.xml out/fx.xml out/ie.xml
```

Implementing the above example with Windows batch files is not very complicated, either. The most important thing to remember is that because `robot` and `rebot` scripts are implemented as batch files on Windows, `call` must be used when running them from another batch file. Otherwise execution would end when the first batch file is finished.

```
@echo off
call robot --variable BROWSER:Firefox --name Firefox --log none --report none --output out\fx.xml login
call robot --variable BROWSER:IE --name IE --log none --report none --output out\ie.xml login
call rebot --name Login --outputdir out --output login.xml out\fx.xml out\ie.xml
```

In the next examples, jar files under the *lib* directory are put into `CLASSPATH` before starting the test execution. In these examples, start-up scripts require that paths to the executed test data are given as arguments. It is also possible to use command line options freely, even though some options have already been set in the script. All this is relatively straight-forward using bash:

```
#!/bin/bash

cp=.
for jar in lib/*.jar; do
    cp=$cp:$jar
done
export CLASSPATH=$cp

robot --ouputdir /tmp/logs --suitestatlevel 2 $*
```

Implementing this using Windows batch files is slightly more complicated. The difficult part is setting the variable containing the needed JARs inside a For loop, because, for some reason, that is not possible without a helper function.

```
@echo off

set CP=.
for %%jar in (lib\*.jar) do (
    call :set_cp %%jar
)
set CLASSPATH=%CP%

robot --ouputdir c:\temp\logs --suitestatlevel 2 %*

goto :eof

:: Helper for setting variables inside a for loop
:set_cp
    set CP=%CP%;%1
goto :eof
```

Modifying Java startup parameters

Sometimes when using Jython there is need to alter the Java startup parameters. The most common use case is increasing the JVM maximum memory size as the default value may not be enough for creating reports and logs when outputs are very big. There are two easy ways to configure JVM options:

1. Set `JYTHON_OPTS` environment variable. This can be done permanently in operating system level or per execution in a custom start-up script.
2. Pass the needed Java parameters with `-J` option to Jython that will pass them forward to Java. This is especially easy when executing installed robot module directly:

```
jython -J-Xmx1024m -m robot tests.robot
```

3.1.7 Debugging problems

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 and in the report file, and sometimes the error message alone is enough to pinpoint the problem. More often that not, however, log files 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 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 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. 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. This functionality is new in Robot Framework 2.9.2.

If the log file still does not have enough information, it is a good idea to enable the syslog 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 keywords *Log* and *Log Variables* are useful. If nothing else works, it is always possible to search help from mailing lists or elsewhere.

Using the Python debugger (pdb)

It is also possible to use the pdb 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 alternativley:

```
Evaluate    pdb.Pdb(stdout=sys.__stdout__).set_trace()    modules=sys, pdb
```

can be used directly in a test case.

3.2 Test execution

This section describes how the test suite structure created from the parsed test data is executed, how to continue executing a test case after failures, and how to stop the whole test execution gracefully.

3.2.1 Execution flow

Executed suites and tests

Test cases are always executed within a test suite. A test suite created from a test case file has tests directly, whereas suites created from 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 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. The exact execution order and how possible setups and teardowns affect the execution are discussed in the following sections.

Setups and teardowns

Setups and teardowns can be used on test suite, test case and user keyword 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 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 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

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 allow listing files nicely one file per line.

It is also possible to randomize the execution order using the `--randomize` option.

Passing execution

Typically test cases, setups and teardowns are considered passed if all keywords they contain are executed and none of them fail. It is also possible to use BuiltIn keywords *Pass Execution* and *Pass Execution If* to stop execution with PASS status and skip the remaining keywords.

How *Pass Execution* and *Pass Execution If* behave in different situations is explained below:

- When used in any setup or teardown (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 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.

Passing execution in the middle of a test, setup or teardown should be used with care. In the worst case it leads to tests that skip all the parts that could actually uncover problems in the tested application. In cases where execution cannot continue do to external factors, it is often safer to fail the test case and make it non-critical.

3.2.2 Continue on failure

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

BuiltIn 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.

Special failures from keywords

Library keywords 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 test library API chapter.

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

BuiltIn 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.

Execution continues on teardowns automatically

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. 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

When using 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.

3.2.3 Stopping test execution gracefully

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 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`

The execution is stopped when `Ctrl-C` is pressed in the console where the tests are running. When running the tests on Python, the execution is stopped immediately, but with Jython it ends only after the currently executing keyword ends.

If `Ctrl-C` is pressed again, the execution ends immediately and reports and logs are not created.

Using signals

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.

Signals have the same limitation on Jython as pressing `Ctrl-C`. Similarly also the second signal stops the execution forcefully.

Using keywords

The execution can be stopped also by the executed keywords. There is a separate *Fatal Error* BuiltIn keyword for this purpose, and custom keywords can use fatal exceptions when they fail.

Stopping when first test case fails

If option `--exitonfailure (-X)` is used, test execution stops immediately if any critical test fails. The remaining tests are marked as failed without actually executing them.

> **Note**
>
> The short option `-X` is new in Robot Framework 3.0.1.

Stopping on parsing or execution error

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, 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

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

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 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

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 and read by Rebot if outputs are post-processed. The mode can also be set when using Rebot if necessary.

3.3.2 Task related command line options

All normal command line options can be used when executing tasks. If there is a need to select only certain tasks for execution, `--task` can be used instead of `--test`. Additionally the aforementioned `--rpa` can be used to control the execution mode.

3.4 Post-processing outputs

XML output files 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

Synopsis

```
rebot [options] robot_outputs
python|jython|ipy|pypy -m robot.rebot [options] robot_outputs
python|jython|ipy|pypy path/to/robot/rebot.py [options] robot_outputs
java -jar robotframework.jar rebot [options] robot_outputs
```

The most common way to use Rebot is using the `rebot` runner script. Alternatively it is possible to execute the installed robot.rebot module or robot/rebot.py file directly using the selected interpreter. The final alternative is using the standalone JAR distribution.

> **Note**
>
> Versions prior to Robot Framework 3.0 installed the `rebot` script only with Python, and used `jyrebot` and `ipyrebot` scripts with Jython and IronPython, respectively. The old interpreter specific scripts were removed in Robot Framework 3.1 and nowadays `rebot` must always be used.

Specifying options and arguments

The basic syntax for using Rebot is exactly the same as when 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 instead of test data files or directories.

Return codes with Rebot

Return codes from Rebot are exactly same as when running tests.

Controlling execution mode

Rebot notices have tests or tasks 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

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. This approach can also work very well if generating reports and logs takes a lot of time when running tests on Jython. Disabling log and report generation and generating them later with Rebot can save a lot of time and use less memory.

3.4.3 Combining outputs

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, 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

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 --critical regression 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

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 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 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. The 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.

Merging suites executed in pieces

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

This section explains different command line options that can be used for configuring the test execution or post-processing outputs. Options related to generated output files are discussed in the next section.

3.5.1 Selecting files to parse

Robot Framework supports test data in various formats, but nowadays the plain text format in dedicated `*.robot` files is the most commonly used. Prior to Robot Framework 3.1, all files in all supported formats were parsed, meaning that also files not containing any test data could be parsed. To avoid parsing non-data files, especially large and slow to parse files, Robot Framework 3.0.1 added the `--extension (-F)` option to select which files to parse. In Robot Framework 3.1 parsing other than `*.robot` files was deprecated and the `--extension` option can be used to explicitly tell the framework to parse other files.

The `--extension` option takes a file extension as an argument, and only files with that extension are parsed. 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.

```
robot --extension robot path/to/tests        # Only parse *.robot files
robot --extension ROBOT:TXT path/to/tests    # Parse *.robot and *.txt files
```

If files in one format use different extensions like `*.rst` and `*.rest`, you need to specify those extensions separately. Using just one of them would mean that other files in that format are skipped.

3.5.2 Selecting test cases

Robot Framework offers several command line options for selecting which test cases to execute. The same options work also when executing tasks and when post-processing outputs with Rebot.

By test suite and test case names

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 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, it is possible to use the `--task` option as an alias for `--test`.

By tag names

It is possible to include and exclude test cases by tag 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 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
```

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

Command line option `--rerunfailed (-R)` can be used to select all failed tests from an earlier 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 using the `--merge` command line option.

Re-executing failed test suites

Command line option `rerunfailedsuites (-S)` can be used to select all failed suites from an earlier 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.

> **Note**
>
> `--rerunfailedsuites` option was added in Robot Framework 3.0.1.

When no tests match selection

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. 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 criticality

The final result of test execution is determined based on critical tests. If a single critical test fails, the whole test run is considered failed. On the other hand, non-critical test cases can fail and the overall status is still considered passed.

All test cases are considered critical by default, but this can be changed with the `--critical (-c)` and `--noncritical (-n)` options. These options specify which tests are critical based on tags, similarly as `--include` and `--exclude` are used to select tests by tags. If only `--critical` is used, test cases with a matching tag are critical. If only `--noncritical` is used, tests without a matching tag are critical. Finally, if both are used, only test with a critical tag but without a non-critical tag are critical.

Both `--critical` and `--noncritical` also support same tag patterns as `--include` and `--exclude`. This means that pattern matching is case, space, and underscore insensitive, `*` and `?` are supported as wildcards, and `AND`, `OR` and `NOT` operators can be used to create combined patterns.

```
--critical regression
--noncritical not_ready
--critical iter-* --critical req-* --noncritical req-6??
```

The most common use case for setting criticality is having test cases that are not ready or test features still under development in the test execution. These tests could also be excluded from the test execution altogether with the `--exclude` option, but including them as non-critical tests enables you to see when they start to pass.

Criticality set when tests are executed is not stored anywhere. If you want to keep same criticality when post-processing outputs with Rebot, you need to use `--critical` and/or `--noncritical` also with it:

```
# Use rebot to create new log and report from the output created during execution
robot --critical regression --outputdir all tests.robot
rebot --name Smoke --include smoke --critical regression --outputdir smoke all/output.xml

# No need to use --critical/--noncritical when no log or report is created
robot --log NONE --report NONE tests.robot
rebot --critical feature1 output.xml
```

3.5.4 Setting metadata

Setting the name

When Robot Framework parses test data, test suite names are created from file and directory names. 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

In addition to defining documentation in the test data, 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.

> **Note**
>
> Prior to Robot Framework 3.1, underscores in the value were converted to spaces same way as with the `--name` option.

Setting free metadata

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. This option may be used several times to set multiple metadata values.

> **Note**
>
> Prior to Robot Framework 3.1, underscores in the value were converted to spaces same way as with the `--name` option.

Setting tags

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.

3.5.5 Configuring where to search libraries and other extensions

When Robot Framework imports a test library, listener, 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. When importing Java based libraries or other extensions on Jython, Java classpath is used in addition to the normal module search path.

Robot Framework uses Python's module search path also when importing 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.

Locations automatically in module search path

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 are automatically installed so that they can be imported without any additional configuration.

`PYTHONPATH`, `JYTHONPATH` and `IRONPYTHONPATH`

Python, Jython and IronPython read additional locations to be added to the module search path from `PYTHONPATH`, `JYTHONPATH` and `IRONPYTHONPATH` environment variables, respectively. 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.

> **Note**
>
> Prior to Robot Framework 2.9, contents of `PYTHONPATH` environment variable were added to the module search path by the framework itself when running on Jython and IronPython. Nowadays that is not done anymore and `JYTHONPATH` and `IRONPYTHONPATH` must be used with these interpreters.

Using `--pythonpath` option

Robot Framework has a separate command line option `--pythonpath (-P)` for adding locations to the module search path. Although the option name has the word Python in it, it works also on Jython and IronPython.

Multiple locations can be given by separating them with a colon, regardless the operating system, or by using this option several times. The given path can also be a glob pattern matching multiple paths, but then it typically needs to be escaped when used on the console.

Examples:

```
--pythonpath libs
--pythonpath /opt/testlibs:mylibs.zip:yourlibs
--pythonpath mylib.jar --pythonpath lib/\*.jar    # '*' is escaped
```

Configuring `sys.path` programmatically

Python interpreters store the module search path they use as a list of strings in sys.path attribute. This list can be updated dynamically during execution, and changes are taken into account next time when something is imported.

Java classpath

When libraries implemented in Java are imported with Jython, they can be either in Jython's normal module search path or in Java classpath. The most common way to alter classpath is setting the `CLASSPATH` environment variable similarly as `PYTHONPATH`, `JYTHONPATH` or `IRONPYTHONPATH`. Alternatively it is possible to use Java's `-cp` command line option. This option is not exposed to the `robot` runner script, but it is possible to use it with Jython by adding `-J` prefix like `jython -J-cp example.jar -m robot.run tests.robot`.

When using the standalone JAR distribution, the classpath has to be set a bit differently, due to the fact that `java -jar` command does support the `CLASSPATH` environment variable nor the `-cp` option. There are two different ways to configure the classpath:

```
java -cp lib/testlibrary.jar:lib/app.jar:robotframework-3.1.jar org.robotframework.RobotFramework tests.robot
java -Xbootclasspath/a:lib/testlibrary.jar:lib/app.jar -jar robotframework-3.1.jar tests.robot
```

3.5.6 Setting variables

Variables can be set from the command line either individually using the `--variable (-v)` option or through 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
```

3.5.7 Dry run

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 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 by adding a special `robot:no-dry-run` keyword tag to them. This is useful if a keyword fails in the dry run mode for some reason, but work fine when executed normally. Disabling the dry run more is a new feature in Robot Framework 3.0.2.

> **Note**
>
> The dry run mode does not validate variables.

3.5.8 Randomizing execution order

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 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
```

3.5.9 Programmatic modification of test data

If the provided built-in features to modify test data before execution are not enough, Robot Framework 2.9 and newer 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, and it possible to modify executed test suites, test cases and keywords 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, 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.

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 first is considered to be the actual separator.

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 can be enabled using the `--prerebotmodifier` option.

Example: Select every Xth test

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, start=0):
        self.x = int(x)
        self.start = int(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, 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
```

Example: Exclude tests by name

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: Skip setups and teardowns

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.keywords.setup = None


class SuiteTeardown(SuiteVisitor):

    def start_suite(self, suite):
        suite.keywords.teardown = None


class TestSetup(SuiteVisitor):

    def start_test(self, test):
        test.keywords.setup = None


class TestTeardown(SuiteVisitor):

    def start_test(self, test):
        test.keywords.teardown = None
```

Assuming that the above modifiers are all in a file named *disable.py* and this file is in the 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
```

3.5.10 Controlling console output

There are various command line options to control how test execution is reported on the console.

Console output type

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 non-critical tests, `F` for failed critical tests, and `x` for tests which are skipped because test execution exit. Failed critical 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.

`none`
No output whatsoever. Useful when creating a custom output using, for example, listeners.

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
```

> **Note**
>
> `--console`, `--dotted` and `--quiet` are new options in Robot Framework 2.9. Prior to that the output was always the same as in the current `verbose` mode.

Console width

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`.

> **Note**
>
> Prior to Robot Framework 2.9 this functionality was enabled with `--monitorwidth` option that was first deprecated and is nowadays removed. The short option `-W` works the same way in all versions.

Console colors

The `--consolecolors (-C)` option is used to control whether colors should be used in the console output. Colors are implemented using ANSI colors except on Windows where, by default, Windows APIs are used instead. Accessing these APIs from Jython is not possible, and as a result colors do not work with Jython on Windows.

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.

> **Note**
>
> Prior to Robot Framework 2.9 this functionality was enabled with `--monitorcolors` option that was first deprecated and is nowadays removed. The short option `-C` works the same way in all versions.

Console markers

Special markers `.` (success) and `F` (failure) are shown on the console when using the verbose output 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.

> **Note**
>
> Prior to Robot Framework 2.9 this functionality was enabled with `--monitormarkers` option that was first deprecated and is nowadays removed. The short option `-K` works the same way in all versions.

3.5.11 Setting listeners

Listeners 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 section for more details about importing listeners and using them in general.

3.6 Created outputs

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

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

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

Output files contain all the test execution results in machine readable XML format. Log, report and xUnit files are typically generated based on them, and they can also be combined and otherwise post-processed with Rebot.

Tip

Generating report and xUnit files as part of test execution does not require processing output files. Disabling log 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. The default name for the output file, when tests are run, is *output.xml*.

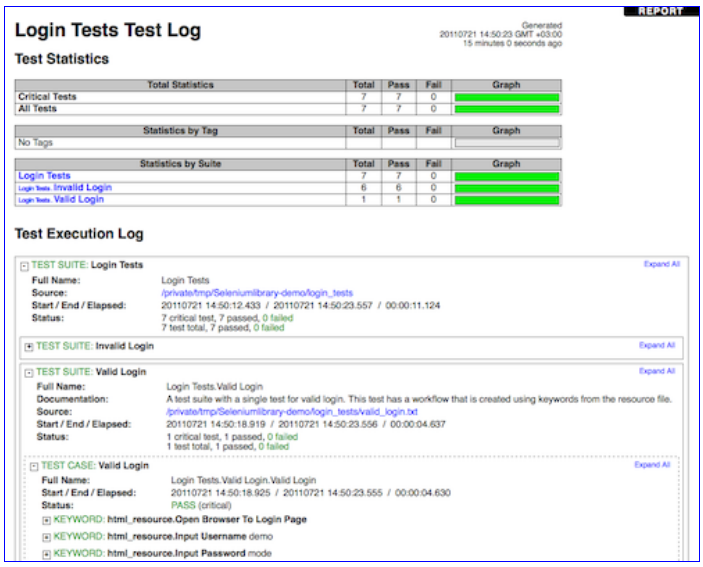
When post-processing outputs with Rebot, new output files are not created unless the `--output` option is explicitly used.

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

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*.



An example of beginning of a log file

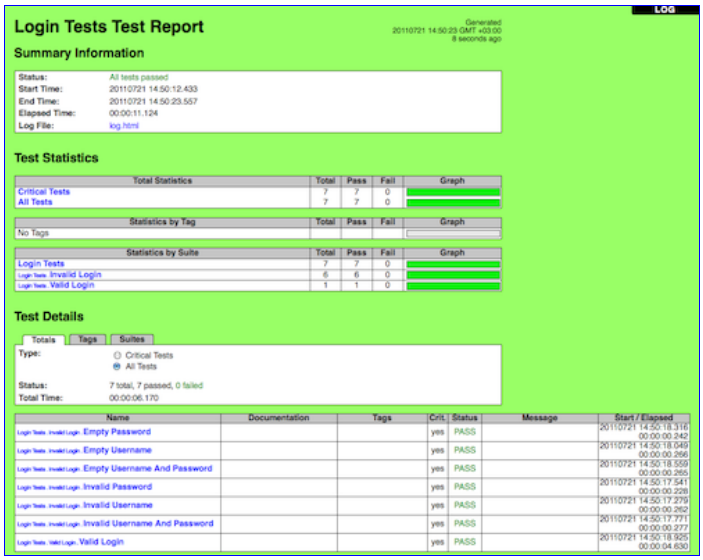
An example of a log file with keyword details visible

Report file

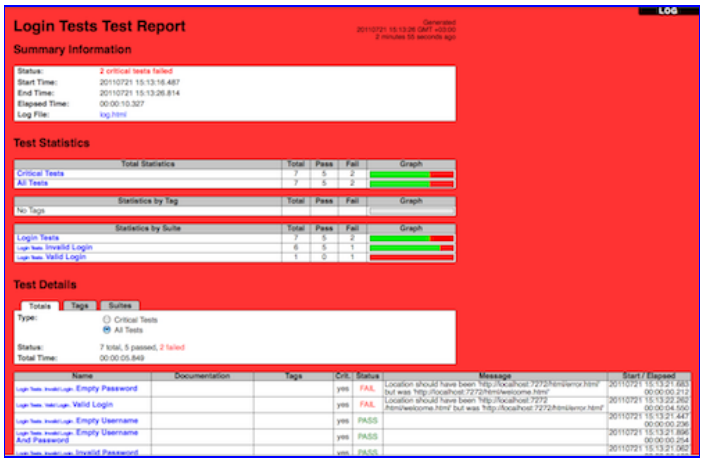
Report files contain an overview of the test execution results in HTML format. They have statistics based on tags and executed test suites, as well as a list of all executed test cases. When both reports and logs are generated, the report has links to the log file for easy navigation to more detailed

information. It is easy to see the overall test execution status from report, because its background color is green, if all critical tests pass, and bright red otherwise.

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*.



An example report file of successful test execution



An example report file of failed test execution

XUnit compatible result file

XUnit result files contain the test execution summary in xUnit compatible XML format. These files can thus be used as an input for external tools that understand xUnit reports. For example, Jenkins continuous integration server supports generating statistics based on xUnit compatible results.

> **Tip**
>
> Jenkins also has a separate 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, as a value.

Because xUnit reports do not have the concept of non-critical tests, all tests in an xUnit report will be marked either passed or failed, with no distinction between critical and non-critical tests. If this is a problem, `--xunitskipnoncritical` option can be used to mark non-critical tests as skipped. Skipped tests will get a message containing the actual status and possible message of the test case in a format like `FAIL: Error message`.

Debug file

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 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

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

The default titles for logs and reports 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

By default the report file has a green background when all the critical tests pass and a red background otherwise. 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 test succeed, the second when only non-critical tests have failed, and the last when there are critical failures. This feature thus allows using a separate background color, for example yellow, when non-critical tests have failed.

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

Available log levels

Messages in log files can have different log levels. Some of the messages are written by Robot Framework itself, but also executed keywords can log 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, 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

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 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 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.

Visible log level

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.

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

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

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

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

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:

```
--tagstatinclude some-tag --tagstatinclude another-tag
--tagstatexclude owner-*
--tagstatinclude prefix-* --tagstatexclude prefix-13
```

Generating combined tag statistics

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 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*
```

owner-*	4	4	0	
smoke & mytag	1	1	0	
smoke NOT owner-janne*	1	1	0	

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

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 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 jython-bug-*:http://bugs.jython.org/issue_%1:Jython-bugs
--tagstatlink owner-*:mailto:%1@domain.com?subject=Acceptance_Tests:Send_Mail
```

jython-bug-1777567	[jython-bugs]	1	1	0	
jython-bug-1778514	[jython-bugs]	1	1	0	
mytag	[Google]	1	1	0	
owner-janne.t.harkonen	[Send mail]	3	3	0	
owner-laukpe	[Send mail]	1	1	0	

Examples of links from tag names

Adding documentation to tags

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 matching multiple tags. `doc` is the assigned documentation. It can contain simple HTML 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

Most of the content of output files 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 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 and when 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

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 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 except the last one.

`WUKS`
: Remove all failing keywords inside BuiltIn 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. The pattern is case, space, and underscore insensitive, and it supports 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 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 and 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 and generating an internal model based on it. Thus it does not reduce memory usage as much as flattening keywords.

> **Note**
>
> `TAG:<pattern>` mode was added in Robot Framework 2.9.

Flattening keywords

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 fully.

`FORITEM`
: Flatten individual for loop iterations.

`NAME:<pattern>`
: Flatten keywords matching the given pattern. Pattern matching rules are same as when removing keywords using `NAME:<pattern>` mode.

`TAG:<pattern>`
: Flatten keywords with tags matching the given pattern. Pattern matching rules are same as when removing keywords using `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 is parsed initially. This can save a significant amount of memory especially with deeply nested keyword structures.

> **Note**
>
> `TAG:<pattern>` mode was added in Robot Framework 2.9.

3.6.6 Setting start and end time of execution

When 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
```

3.6.7 Limiting error message length in reports

If a test case fails and has a long error message, the message shown in reports 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 as messages of the failed keywords.

> **Note**
>
> The `--maxerrorlines` option is new in Robot Framework 3.1.

3.6.8 Programmatic modification of results

If the provided built-in features to modify results are are not enough, Robot Framework 2.9 and newer 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 that can be enabled with the `--prerunmodifier` option. The obvious difference is that this time modifiers operate with the result model, not the 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):
        self.max_milliseconds = float(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.

3.6.9 System log

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 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 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

4.1 Creating test libraries

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

Supported programming languages

Robot Framework itself is written with Python and naturally test libraries extending it can be implemented using the same language. When running the framework on Jython, libraries can also be implemented using Java. Pure Python code works both on Python and Jython, assuming that it does not use syntax or modules that are not available on Jython. When using Python, it is also possible to implement libraries with C using Python C API, although it is often easier to interact with C code from Python libraries using ctypes module.

Libraries implemented using these natively supported languages can also act as wrappers to functionality implemented using other programming languages. A good example of this approach is the Remote library, and another widely used approaches is running external scripts or tools as separate processes.

Different test library APIs

Robot Framework has three different test library APIs.

Static API

The simplest approach is having a module (in Python) or a class (in Python or Java) with methods which map directly to keyword names. Keywords also take the same arguments as the methods implementing them. Keywords report failures with exceptions, log by writing to standard output and can return 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 and the hybrid library API differ from it is then discussed in sections of their own.

The examples in this chapter are mainly about using Python, but they should be easy to understand also for Java-only developers. In those few cases where APIs have differences, both usages are explained with adequate examples.

4.1.2 Creating test library class or module

Test libraries can be implemented as Python modules and Python or Java classes.

Test library names

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*. Similarly, a Java class `YourLibrary`, when it is not in any package, creates a library with exactly that name.

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*.

Java classes in a non-default package must be taken into use with the full name. For example, class `MyLib` in `com.mycompany.myproject` package must be imported with name *com.mycompany.myproject.MyLib*.

> **Tip**
>
> If the library name is really long, for example when the Java package name is long, it is recommended to give the library a simpler alias by using the WITH NAME syntax.

Providing arguments to test libraries

All test libraries implemented as classes can take arguments. These arguments are specified in the Setting table 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, with the exception that there is no variable argument support for Java libraries. 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, first one in Python and second in Java, 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)
```

```
public class AnotherLib {

    private String setting = null;

    public AnotherLib(String setting) {
        setting = setting;
    }

    public void doSomething() {
        if setting.equals("42") {
            // do something ...
        }
    }
}
```

Test library scope

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 CASE`
: A new instance is created for every test case. A possible suite setup and suite teardown share yet another instance. This is the default.

`TEST 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.

`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, a new instance is created every time regardless the scope.

When the `TEST SUITE` or `GLOBAL` scopes are used with test 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 Python library using the `TEST SUITE` scope:

```
class ExampleLibrary:

    ROBOT_LIBRARY_SCOPE = 'TEST SUITE'

    def __init__(self):
        self._counter = 0

    def count(self):
        self._counter += 1
        print(self._counter)

    def clear_counter(self):
        self._counter = 0
```

Example Java library using the `GLOBAL` scope:

```
public class ExampleLibrary {
```

```
    public static final String ROBOT_LIBRARY_SCOPE = "GLOBAL";

    private int counter = 0;

    public void count() {
        counter += 1;
        System.out.println(counter);
    }

    public void clearCounter() {
        counter = 0;
    }
}
```

Specifying library version

When a test library is taken into use, Robot Framework tries to determine its version. This information is then written into the syslog to provide debugging information. Library documentation tool Libdoc also writes this information into the keyword documentations it generates.

Version information is read from attribute `ROBOT_LIBRARY_VERSION`, similarly as test 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. For Java libraries the version attribute must be declared as `static final`.

An example Python module using `__version__`:

```
__version__ = '0.1'

def keyword():
    pass
```

A Java class using `ROBOT_LIBRARY_VERSION`:

```
public class VersionExample {

    public static final String ROBOT_LIBRARY_VERSION = "1.0.2";

    public void keyword() {
    }
}
```

Specifying documentation format

Library documentation tool Libdoc supports documentation in multiple formats. If you want to use something else than Robot Framework's own documentation formatting, you can specify the format in the source code using `ROBOT_LIBRARY_DOC_FORMAT` attribute similarly as scope and 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). Using the `reST` format requires the docutils module to be installed when documentation is generated.

Setting the documentation format is illustrated by the following Python and Java examples that use reStructuredText and HTML formats, respectively. See Documenting libraries section and 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
```

```
/**
 * A library for <i>documentation format</i> demonstration purposes.
 *
 * This documentation is created using <a href="http://www.w3.org/html">HTML</a>.
 * Here is a link to the only `Keyword`.
 */
public class DocFormatExample {
```

```
    public static final String ROBOT_LIBRARY_DOC_FORMAT = "HTML";

    /**<b>Nothing</b> to see here. Not even in the table below.
     *
     * <table>
     * <tr><td>Table</td><td>here</td><td>has</td></tr>
     * <tr><td>nothing</td><td>to</td><td>see.</td></tr>
     * </table>
     */
    public void keyword() {
    }
}
```

Library acting as listener

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 Test libraries as listeners section.

4.1.3 Creating static keywords

What methods are considered keywords

When the static library API is used, Robot Framework uses reflection to find out what public methods the library class or module contains. It will exclude all methods starting with an underscore (unless using a custom keyword name), and with Java libraries also methods implemented only in the implicit base class `java.lang.Object` are excluded. All the methods that are not ignored are considered keywords. For example, the Python and Java libraries below implement 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()
```

```
public class MyLibrary {

    public String myKeyword(String arg) {
        return helperMethod(arg);
    }

    private String helperMethod(String arg) {
        return arg.toUpperCase();
    }
}
```

When implementing a library as a Python or Java class, also methods in possible base classes are considered keywords. When implementing a library as a Python module, also possible functions imported into the module namespace become keywords. For example, if the module below would be used as a library, it would contain keywords *Example Keyword*, *Second Example* and also *Current Thread*.

```
from threading import current_thread


def example_keyword():
    print('Running in thread "%s".' % current_thread().name)

def second_example():
    pass
```

A simple way to avoid imported functions becoming keywords is to only import modules (e.g. `import threading`) and 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. If it is used, only the listed functions can be keywords. For example, the library below implements only keywords *Example Keyword* and *Second Example*.

```
from threading import current_thread


__all__ = ['example_keyword', 'second_example']


def example_keyword():
    print('Running in thread "%s".' % current_thread().name)
```

```
def second_example():
    pass

def not_exposed_as_keyword():
    pass
```

Keyword names

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 Python library implemented as a module in the *MyLibrary.py* file:

```
def hello(name):
    print("Hello, %s!" % name)

def do_nothing():
    pass
```

Example Java library implemented as a class in the *MyLibrary.java* file:

```
public class MyLibrary {

    public void hello(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public void doNothing() {
    }

}
```

The example below illustrates how the example libraries above can be used. If you want to try this yourself, make sure that the library is in the module search path.

```
*** Settings ***
Library    MyLibrary

*** Test Cases ***
My Test
    Do Nothing
    Hello    world
```

Using a custom keyword 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. This is typically easiest done by using the `robot.api.deco.keyword` decorator as follows:

```
from robot.api.deco import keyword

@keyword('Login Via User Panel')
def login(username, password):
    # ...
```

```
*** Test Cases ***
My Test
    Login Via User Panel    ${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 without actually changing keyword names. Starting from Robot Framework 3.0.2, 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 Embedded Arguments syntax.

Keyword tags

Starting from Robot Framework 2.9, library keywords and user keywords can have tags. Library keywords can define them by setting the `robot_tags` attribute on the method to a list of desired tags. The `robot.api.deco.keyword` decorator may be used as a shortcut for setting this attribute when used as follows:

```
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 with `Tags:` prefix and separated by a comma. For example:

```
def login(username, password):
    """Log user in to SUT.

    Tags: tag1, tag2
    """
    # ...
```

Keyword arguments

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 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, both the Python and Java methods 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 Python 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))
```

> **Note**
>
> A major limitation with Java libraries using the static library API is that they do not support the named argument syntax. If this is a blocker, it is possible to either use Python or switch to the dynamic library API.

Default values to keywords

It is often useful that some of the arguments that a keyword uses have default values. Python and Java have different syntax for handling default values to methods, and the natural syntax of these languages can be used when creating test libraries for Robot Framework.

Default values with Python

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
```

Default values with Java

In Java one method can have several implementations with different signatures. Robot Framework regards all these implementations as one keyword, which can be used with different arguments. This syntax can thus be used to provide support for the default values. This is illustrated by the example below, which is functionally identical to the earlier Python example:

```
public void oneDefault(String arg) {
    System.out.println("Argument has value " + arg);
}

public void oneDefault() {
    oneDefault("default");
}

public void multipleDefaults(String arg1, String arg2, String arg3) {
    System.out.println("Got arguments " + arg1 + ", " + arg2 + " and " + arg3);
}

public void multipleDefaults(String arg1, String arg2) {
    multipleDefaults(arg1, arg2, "default 2");
}

public void multipleDefaults(String arg1) {
    multipleDefaults(arg1, "default 1");
}
```

Variable number of arguments (`*varargs`)

Robot Framework supports also keywords that take any number of arguments. Similarly as with the default values, the actual syntax to use in test libraries is different in Python and Java.

Variable number of arguments with Python

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\nOthers:" % 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
```

Variable number of arguments with Java

Robot Framework supports Java varargs syntax for defining variable number of arguments. For example, the following two keywords are functionally identical to the above Python examples with same names:

```
public void anyArguments(String... varargs) {
    System.out.println("Got arguments:");
    for (String arg: varargs) {
        System.out.println(arg);
    }
}

public void oneRequired(String required, String... others) {
    System.out.println("Required: " + required + "\nOthers:");
    for (String arg: others) {
        System.out.println(arg);
    }
}
```

It is also possible to use variable number of arguments also by having an array or `java.util.List` as the last argument, or second to last if free keyword arguments (**kwargs) are used. This is illustrated by the following examples that are functionally identical to the previous ones:

```
public void anyArguments(String[] varargs) {
    System.out.println("Got arguments:");
    for (String arg: varargs) {
        System.out.println(arg);
    }
}

public void oneRequired(String required, List<String> others) {
```

```
    System.out.println("Required: " + required + "\nOthers:");
    for (String arg: others) {
        System.out.println(arg);
    }
}
```

Note

Only `java.util.List` is supported as varargs, not any of its sub types.

The support for variable number of arguments with Java keywords has one limitation: it works only when methods have one signature. Thus it is not possible to have Java keywords with both default values and varargs.

Free keyword arguments (`**kwargs`)

Robot Framework supports Python's **kwargs syntax and extends that support also to Java. How to use use keywords that accept *free keyword arguments*, also known as *free named arguments*, is discussed under the Creating test cases section. In this section we take a look at how to create such keywords using Python and Java.

Free keyword arguments with Python

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 `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 like `foo\=quux`.

The following example illustrates how normal arguments, varargs, and kwargs work together:

```
def various_args(arg, *varargs, **kwargs):
    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 library.

Free keyword arguments with Java

Also Java libraries support the free keyword arguments syntax. Java itself has no kwargs syntax, but keywords can have `java.util.Map` as the last argument to specify that they accept kwargs.

If a Java keyword accepts kwargs, Robot Framework will automatically pack all arguments in `name=value` syntax at the end of the keyword call into a `Map` and pass it to the keyword. For example, following example keywords can be used exactly like the previous Python examples:

```
public void exampleKeyword(Map<String, String> stuff):
    for (String key: stuff.keySet())
        System.out.println(key + " " + stuff.get(key));

public void variousArgs(String arg, List<String> varargs, Map<String, Object> kwargs):
```

```
    System.out.println("arg: " + arg);
    for (String varg: varargs)
        System.out.println("vararg: " + varg);
    for (String key: kwargs.keySet())
        System.out.println("kwarg: " + key + " " + kwargs.get(key));
```

Note

The type of the kwargs argument must be exactly `java.util.Map`, not any of its sub types.

Note

Similarly as with the varargs support, a keyword supporting kwargs cannot have more than one signature.

Keyword-only arguments

Starting from Robot Framework 3.1, it is possible to use named-only arguments with different keywords. When implementing libraries using Python, this support is provided by Python's keyword-only arguments:

```
def sort_words(*words, case_sensitive=False):
    key = str.lower if case_sensitive else None
    return sorted(words, key=key)
```

```
*** Test Cases ***
Example
    Sort Words    Foo    bar    baZ
    Sort Words    Foo    bar    baZ    case_sensitive=True
```

Due to keyword-only arguments being a Python 3 feature, libraries using Python 2 cannot use it. Time to upgrade!

Argument types

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 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 to other types.
- It is possible to specify argument types explicitly using Python 3 function annotations or the @keyword decorator. In these cases Robot Framework converts arguments automatically.
- Automatic conversion is also done based on keyword default values.
- Arguments to Java keywords are converted based on argument type information.

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 section specifies which argument conversion are supported in these cases.

Manual argument conversion

If no type information is specified to Robot Framework, all arguments not passed as 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.

Specifying argument types using function annotations

Starting from Robot Framework 3.1, arguments passed to keywords as strings are automatically

converted if argument type information is available. The most natural way to specify types is using Python 3 function annotations. 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 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.

> **Note**
>
> Because function annotations are a Python 3 feature, using them in a library that should also work with Python 2 is not possible.

> **Note**
>
> Using function annotations with Robot Framework 3.0.2 or earlier is not possible at all.

Specifying argument types using `@keyword` decorator

An alternative way to specify explicit argument types is using the `robot.api.deco.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:

```
@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` or any other non-true value 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 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 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 argument conversion based on default values is not desired with a certain argument, it can be disabled by specifying a type for that argument explicitly. Alternatively argument conversion can be disabled altogether with the @keyword decorator like `@keyword(types=None)`.

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 or the @keyword decorator.
- If not explicitly specified, type can be got implicitly from argument default values.
- Conversion is done only if the argument that is used is itself a Unicode string.
- With most of the types failed conversion causes an error if the type has been specified explicitly. Exceptions to this rule are mentioned in the table below.
- With most of the types string `NONE`, case-insensitively, is converted to Python `None`. Exceptions are mentioned in the table below.

The type to use can be specified either using concrete types (e.g. list), by using Abstract Base Classes (ABC) (e.g. Sequence), or by using sub classes of these types (e.g. MutableSequence). In all these cases the argument is converted to the concrete type.

Also types in in the typing 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.

Supported argument conversions

Type	ABC	Aliases	Explanation	Examples
bool		boolean	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 are passed as-is, allowing keywords to handle them specially if needed. All comparisons are case-insensitive.	`TRUE` (converted to `True`) `off` (converted to `False`) `foobar` (returned as-is)
int	Integral	integer, long	Conversion is done using the int built-in function. If that fails and type is got implicitly from default values, also float conversion is attempted.	`42` `3.14` (only with implicit type)
float	Real	double	Conversion is done using the float built-in.	`3.14` `2.9979e8`
Decimal			Conversion is done using the Decimal class.	`3.14`
bytes	ByteString		Argument is converted to bytes so that each Unicode code point below 256 is directly mapped to a matching byte. Higher code points are not allowed. String `NONE` (case-insensitively) is converted to matching bytes, not to Python `None`. When using Python 2, byte conversion is only done if type is specified explicitly.	`foobar` `hyvä` (converted to `hyv\xe4`) `\x00` (the null byte)
bytearray			Same conversion as with bytes but the result is a bytearray.	

datetime			Argument is expected to be a timestamp in 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.	`2018-09-12T15:47:05.123456`
				`2018-09-12 15:47`
				`2018-09-12`

date			Same conversion as with datetime but all time components are expected to be omitted or to be zeros.	`2018-09-12`
timedelta			String is expected to represent a time interval in one of the time formats Robot Framework supports: time as number, time as time string or time as "timer" string.	`42` (42 seconds) `1 minute 2 seconds` `01:02` (same as above)

Enum			The specified type must be an enumeration (a subclass of Enum) and arguments themselves must match its members.	`class Color(Enum):`
				`  RED = 1`
				`  GREEN = 2`
				
				`GREEN`

NoneType			String `NONE` (case-insensitively) is converted to `None` object, other values are passed as-is. Mainly relevant when type is got implicitly from `None` being a	`None`

			default value.	
list	Sequence		Argument must be be a Python list literal. It is converted to an actual list using the ast.literal_eval function. The list can contain any values ast.literal_eval supports inside it, including other lists or other containers.	`['foo', 'bar']` `[('one', 1), ('two', 2)]`
tuple			Same as list but the argument must be a tuple literal.	`('foo', 'bar')`
dict	Mapping	dictionary, map	Same as list but the argument must be a dictionary literal.	`{'a': 1, 'b': 2}` `{'key': 1, 'nested': {'key': 2}}`
set	Set		Same as list but the argument must be a set literal or `set()` to create an empty set. Not supported on Python 2.	`{1, 2, 3, 42}` `set()`
frozenset			Same conversion as with set but the result is a frozenset.	

Argument types with Java

Arguments to Java methods have types, and all the base types are handled automatically. This means that arguments that are normal strings in the test data are coerced to correct type at runtime. The types that can be coerced are:

- integer types (`byte`, `short`, `int`, `long`)
- floating point types (`float` and `double`)
- the `boolean` type
- object versions of the above types e.g. `java.lang.Integer`

The coercion is done for arguments that have the same or compatible type across all the signatures of the keyword method. In the following example, the conversion can be done for keywords `doubleArgument` and `compatibleTypes`, but not for `conflictingTypes`.

```
public void doubleArgument(double arg) {}

public void compatibleTypes(String arg1, Integer arg2) {}
public void compatibleTypes(String arg2, Integer arg2, Boolean arg3) {}

public void conflictingTypes(String arg1, int arg2) {}
public void conflictingTypes(int arg1, String arg2) {}
```

The coercion works with the numeric types if the test data has a string containing a number, and with the boolean type the data must contain either string `true` or `false`. Coercion is only done if the original value was a string from the test data, but it is of course still possible to use variables containing correct types with these keywords. Using variables is the only option if keywords have conflicting signatures.

```
*** Test Cases ***
Coercion
    Double Argument     3.14
    Double Argument     2e16
    Compatible Types    Hello, world!    1234
    Compatible Types    Hi again!    -10    true

No Coercion
    Double Argument    ${3.14}
    Conflicting Types    1       ${2}    # must use variables
    Conflicting Types    ${1}    2
```

Argument type coercion works also with Java library constructors.

> **Note**
>
> Converting arguments passed to Java based keywords is an old feature and independent on the support to convert arguments of Python keywords in Robot Framework 3.1 and newer. Conversion functionality may be unified in the future.

Using decorators

When writing static keywords, it is sometimes useful to modify them with Python's decorators. However, decorators modify function signatures, and can confuse Robot Framework's introspection when determining which arguments keywords accept. This is especially problematic when creating library documentation with Libdoc and when using RIDE. To avoid this issue, either do not use decorators, or use the handy decorator module to create signature-preserving decorators.

Embedding arguments into keyword names

Library keywords can also accept arguments which are passed using Embedded Argument syntax. The `robot.api.deco.keyword` decorator can be used to create a custom keyword 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 type conversion works if type information is specified somehow. With Python 3 it is convenient to use function annotations, and alternatively it is possible to pass types to the @keyword decorator:

```
@keyword('Add ${quantity:\d+} copies of ${item} to cart',
         types={'quantity': int})
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

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

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`.

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
```

Java:

```
public class MyError extends RuntimeException {
    public static final boolean 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.

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. 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`.

Stopping test execution

It is possible to fail a test case so that the whole test execution is stopped. This is done simply by having a special `ROBOT_EXIT_ON_FAILURE` attribute with `True` value set on the exception raised from the keyword. This is illustrated in the examples below.

Python:

```
class MyFatalError(RuntimeError):
    ROBOT_EXIT_ON_FAILURE = True
```

Java:

```
public class MyFatalError extends RuntimeException {
    public static final boolean ROBOT_EXIT_ON_FAILURE = true;
}
```

Continuing test execution despite of failures

It is possible to continue test execution even when there are failures. The way to signal this from test libraries is adding a special `ROBOT_CONTINUE_ON_FAILURE` attribute with `True` value to the exception used to communicate the failure. This is demonstrated by the examples below.

Python:

```
class MyContinuableError(RuntimeError):
    ROBOT_CONTINUE_ON_FAILURE = True
```

Java:

```
public class MyContinuableError extends RuntimeException {
    public static final boolean ROBOT_CONTINUE_ON_FAILURE = true;
}
```

Logging information

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 simply by writing to the standard output stream (stdout) or to the standard error stream (stderr), and they can even use different log levels. Another, and often better, logging possibility is using the 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 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.

> **Note**
>
> In Robot Framework 2.9, new functionality was added to automatically add ERRORs logged by keywords to the Test Execution Errors section.

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, 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 and it must be placed after the log level 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 both using Python and Java. If you are using Python, it is, however, even easier to get accurate timestamps using the programmatic logging APIs. A big benefit of adding timestamps explicitly is that this approach works also with the remote library interface.

Python:

```
import time

def example_keyword():
    print('*INFO:%d* Message with timestamp' % (time.time()*1000))
```

Java:

```
public void exampleKeyword() {
    System.out.println("*INFO:" + System.currentTimeMillis() + "* Message with timestamp");
}
```

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. A bonus is that these approaches work both with Python and Java based libraries.

Another option, that is only available with Python, 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:

```
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. Java programmers should regard the code `print('message')` as pseudocode meaning `System.out.println("message");`.

```
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
```

Programmatic logging APIs

Programmatic APIs provide somewhat cleaner way to log information than using the standard output and error streams. Currently these interfaces are available only to Python bases test libraries.

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.

The public logging API is thoroughly documented as part of the API documentation at 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 module.

Using Python's standard `logging` module

In addition to the new public logging API, Robot Framework offers a built-in support to Python's standard logging 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, 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

Libraries can also log during the test library import and initialization. These messages do not appear in the log file like the normal log messages, but are instead written to the syslog. 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 section in the log file.

Logging during the import and initialization is possible both using the standard output and error streams and the programmatic logging APIs. Both of these are demonstrated below.

Java library logging via stdout during initialization:

```
public class LoggingDuringInitialization {

    public LoggingDuringInitialization() {
        System.out.println("*INFO* Initializing library");
    }

    public void keyword() {
```

```
        // ...
    }
}
```

Python 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__` or in Java constructor, the messages may be logged multiple times depending on the test library scope.

Returning values

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 in the test data and then used as inputs for other keywords, even from different test libraries.

Values are returned using the `return` statement both from the Python and Java methods. Normally, one value is assigned into one scalar variable, as illustrated in the example below. This example also illustrates that it is possible to return any objects and to use 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 at once, into a list variable, or into scalar variables and a list variable. All these usages require that returned values are Python lists or tuples or in Java arrays, Lists, or Iterators.

```
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

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 are silently ignored.

There is also a `BackgroundLogger` in separate 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

Documenting libraries

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 with Python and Javadoc with Java, as in the examples 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
```

```
/**
 *  This is an example library with some documentation.
 */
public class MyLibrary {

    /**
     * This keyword has only a short documentation
     */
    public void keywordWithShortDocumentation(String argument) {
    }

    /**
     * First line of the documentation is here.
     *
     * Longer documentation continues here and it can contain
     * multiple lines or paragraphs.
     */
    public void keywordWithLongerDocumentation() {
    }

}
```

Both Python and Java have 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. This tool can create a library documentation from both Python and Java libraries using the static library API, such as the ones above, but it also handles libraries using the dynamic library API and 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 (for example, as a tool tip) and also shown in the test logs. The latter does not work with Java libraries using the static API, though, because their documentation is not available at runtime.

By default documentation is considered to follow Robot Framework's 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 formats. See Specifying documentation format section for information how to set the format in the library source code and 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.

> **Note**
>
> If you want to use non-ASCII characters in the documentation of Python libraries, you must either use UTF-8 as your source code encoding or create docstrings as Unicode. When using Python 3, UTF-8 is the default source encoding.

Testing libraries

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.

Both Python and Java have 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 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

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 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 or other tools, it is automatically in the module search path.

When using Java, it is natural to package libraries into a JAR archive. The JAR package must be put into the module search path before running tests, but it is easy to create a start-up script that does that automatically.

Deprecating keywords

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. The deprecation warning starts with text `Keyword '<name>' is deprecated.` and has rest of the short documentation 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. The only exception are keywords implemented in a Java test library that uses the static library interface because their documentation is not available at runtime. With such keywords, it possible to use user keywords as wrappers and deprecate them.

> **Note**
>
> Prior to Robot Framework 2.9 the documentation must start with `*DEPRECATED*` exactly without any extra content before the closing `*`.

4.1.6 Dynamic library API

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 (except for the documentation of Java libraries), 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.

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. However, if you use Java, you may want to examine the JavaLibCore project before implementing your own system. This collection of reusable tools supports several ways of creating keywords, and it is likely that it already has a mechanism that suites your needs. Python users may also find the similar PythonLibCore project useful.

Getting keyword names

Dynamic libraries tell what keywords they implement with the `get_keyword_names` method. The method also has the alias `getKeywordNames` that is recommended when using Java. 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 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`. See Using a custom keyword name for more about this decorator.

```
from robot.api.deco import keyword

class DynamicExample:

    def get_keyword_names(self):
        return [name for name in dir(self) if hasattr(getattr(self, name), 'robot_name')]

    def helper_method(self):
        # ...

    @keyword
    def keyword_method(self):
        # ...
```

Running keywords

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 given to the keyword in the test data, and the optional third argument is a dictionary (map in Java) containing named arguments. If the third argument is missing, free named arguments and named-only arguments 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 implemented in Python.

```
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

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, have default values, accept variable number of arguments, accept free named arguments and have 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. Note that the examples use Python syntax for lists, but Java developers should use Java lists or String arrays instead.

Representing different arguments with `get_keyword_arguments`

Expected arguments	How to represent	Examples
No arguments	Empty list.	`[]`
One or more positional argument	List of strings containing argument names.	`['argument']`, `['arg1', 'arg2', 'arg3']`
Default values for arguments	Default values separated from argument names with `=`. Default values are always considered to be strings.	`['arg=default value']`, `['a', 'b=1', 'c=2']`
Variable number of arguments (varargs) (varargs)	Argument after possible positional arguments and their defaults has `*` prefix.	`['*varargs']`, `['argument', '*rest']`, `['a', 'b=42', '*c']`
Free named arguments (kwargs)	Last arguments has `**` prefix. Requires `run_keyword` to support free named arguments.	`['**named']`, `['a', 'b=42', '**c']`, `['*varargs', '**kwargs']`
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. 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 and the Libdoc tool needs them to be able to create a meaningful library documentation.

If `get_keyword_arguments` is missing or returns Python `None` or Java `null` 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 or not.

Getting keyword argument types

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, 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. 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 section for more information about supported types and how to specify them.

Getting keyword tags

Starting from Robot Framework 3.0.2, dynamic libraries can report 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`. This approach works also with Robot Framework versions prior to 3.0.2.

> **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

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 implemented with Python. The main use case is getting keywords' documentations into a library documentation generated by Libdoc. Additionally, the first line of the documentation (until the first `\n`) is shown in test logs.

Getting general library documentation

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 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 in practice.

Python based 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.

Named argument syntax with dynamic libraries

Also the dynamic library API supports the named argument syntax. Using the syntax works based on the argument names and default values got from the library 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

Dynamic libraries can also support free named arguments (`**named`). A mandatory precondition for this support is that the `run_keyword` method takes three arguments: 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. 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

Starting from Robot Framework 3.1, dynamic libraries can have named-only arguments. This requires that the `run_keyword` method takes three arguments: the third getting the named-only arguments along with the other named arguments.

In the argument specification 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

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 of the implemented keywords.
`run_keyword`	`name, arguments, kwargs`	Execute the specified keyword with given arguments. `kwargs` is optional.
`get_keyword_arguments`	`name`	Return keywords' argument specification. Optional method.
`get_keyword_types`	`name`	Return keywords' argument type information. Optional method. New in RF 3.1.
`get_keyword_tags`	`name`	Return keywords' tags. Optional method. New in RF 3.0.2.
`get_keyword_documentation`	`name`	Return keywords' and library's documentation. Optional method.

It is possible to write a formal interface specification in Java as below. However, remember that libraries *do not need* to implement any explicit interface, because Robot Framework directly checks with reflection if the library has the required `get_keyword_names` and `run_keyword` methods or their camelCase aliases.

```
public interface RobotFrameworkDynamicAPI {

    List<String> getKeywordNames();

    Object runKeyword(String name, List arguments);

    Object runKeyword(String name, List arguments, Map kwargs);

    List<String> getKeywordArguments(String name);

    List<String> getKeywordTypes(String name);

    List<String> getKeywordTags(String name);

    String getKeywordDocumentation(String name);

}
```

> **Note**
>
> In addition to using `List`, it is possible to use also arrays like `Object[]` or `String[]`.

A good example of using the dynamic API is Robot Framework's own Remote library.

4.1.7 Hybrid library API

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

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

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 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`.

The hybrid API is not very useful with Java, because it is not possible to handle missing methods with it. Of course, it is possible to implement all the methods in the library class, but that brings few benefits compared to the static API.

Getting keyword arguments and documentation

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

When implementing a test library in Python, 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 when using Python. 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 library.

4.1.8 Using Robot Framework's internal modules

Test libraries implemented with Python 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

API documentation is hosted separately at the excellent Read the Docs service. If you are unsure how to use certain API or is using them forward compatible, please send a question to mailing list.

Using BuiltIn library

The safest API to use are methods implementing keywords in the BuiltIn 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. 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

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

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

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. This example uses Python, but you can obviously extend an existing Java library in Java code the same way.

```
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. 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

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.

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

BuiltIn 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. The following example illustrates how to implement the same *Title Should Start With* keyword as in the earlier example about 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

Test libraries that use the dynamic or 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

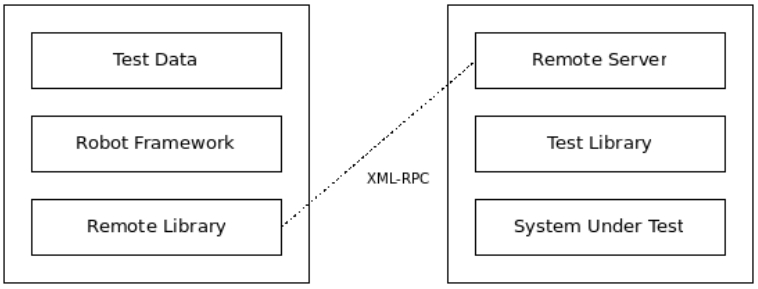
The remote library interface provides means for having test libraries on different machines than where Robot Framework itself is running, and also for implementing libraries using other languages than the natively supported Python and Java. For a test library, user remote libraries look pretty much the same as any other test library, and developing test libraries using the remote library interface is also very close to creating normal test libraries.

4.2.1 Introduction

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 protocol. There exists ready-made generic 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. 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 on top of an XML-RPC channel. The high level architecture of all this is illustrated in the picture below:



> **Note**
>
> The remote client uses Python's standard XML-RPC module. It does not support custom XML-RPC extensions implemented by some XML-RPC servers.

4.2.2 Putting Remote library to use

Importing Remote library

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.

```
*** 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 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. Setting a custom timeout does not work with IronPython.

> **Note**
>
> Port `8270` is the default port that remote servers are expected to use and it has been registered by IANA 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.

> **Note**
>
> If the URI contains no path after the server address, the XML-RPC module 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

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 or SSH libraries, can start the server up, but then you may need to use 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

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 like `${result.key}`. This works also with nested dictionaries like `${root.child.leaf}`. New functionality in Robot Framework 2.9.
- Strings containing bytes in the ASCII range that cannot be represented in XML (e.g. the null byte) are sent as 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

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 provided Python and Ruby servers can also be used as examples.

The remote protocol is implemented on top of XML-RPC, 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.

Required methods

A remote server is an XML-RPC server that must have the same methods in its public interface as the dynamic library API has. Only `get_keyword_names` and `run_keyword` are actually required, but `get_keyword_arguments`, `get_keyword_types`, `get_keyword_tags` and `get_keyword_documentation` are also recommended. Notice that using the camelCase format like `getKeywordNames` in method names is not possible similarly as in the normal dynamic API. 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 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.

The Python remote server can be used as a reference implementation.

Getting remote keyword names and other information

The Remote library gets the list of keywords that a remote server provides by using the `get_keyword_names` method. Remote servers must implement this method and the method must return keyword names 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. Arguments must be returned as a list of strings in the same format as with dynamic libraries, tags as a list of strings, and documentation as a string.

Type information 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 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, and the empty string should be used instead if a marker telling certain argument does not have type information is needed.

Remote servers can also provide general library documentation to be used when generating documentation with the Libdoc tool.

> **Note**
>
> `get_keyword_tags` is new in Robot Framework 3.0.2. With earlier versions keyword tags can be embedded into the keyword documentation.

> **Note**
>
> `get_keyword_types` is new in Robot Framework 3.1.

Executing remote keywords

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. Base types can be used as arguments directly, but more complex types are converted to supported 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 in format `*INFO* First message\n*HTML* <b>2nd</b>\n*WARN* Another message`. It is also possible to embed timestamps to the log messages like `*INFO:1308435758660* Message with timestamp`.
return	Possible return value. Must be one of the supported types.
error	Possible error message. Used only when the execution fails.
traceback	Possible stack trace to write into the log file 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.
fatal	Like `continuable`, but denotes that the occurred failure is fatal.

Different argument syntaxes

The Remote library is a dynamic library, and in general it handles different argument syntaxes according to the same rules as any other dynamic library. This includes mandatory arguments, default values, varargs, as well as named argument syntax.

Also free named arguments (`**kwargs`) works mostly the same way as with other 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.

```python
def run_keyword(name, args, kwargs=None):
    # ...
```

```java
public Map run_keyword(String name, List args) {
    // ...
}

public Map run_keyword(String name, List args, Map kwargs) {
    // ...
}
```

4.3 Listener interface

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, and they can be implemented both with Python and Java. 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 that receive notifications while that library is active.

4.3.1 Taking listeners into use

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 interface, similarly as test library names are got from classes implementing them. The specified listeners must be in the same 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. 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:\data;E:\extra tests.robot
```

4.3.2 Listener interface versions

There are two supported listener interface versions. Listener version 2 has been available since Robot Framework 2.1, and version 3 is supported by Robot Framework 3.0 and newer. A listener must have 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. There has also been an older listener version 1, but it is not supported anymore by Robot Framework 3.0.

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 for more information about what listeners can do.

Another difference between versions 2 and 3 is that the former supports both Python and Java but the latter supports only Python.

4.3.3 Listener interface methods

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

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 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 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` or `FAIL`. • `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. • `doc`: Test documentation. • `tags`: Test tags as a list of strings. • `critical`: `yes` or `no` depending is test considered critical or not. • `template`: The name of the template used for the test. An empty string if the test not templated. • `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`. • `doc`: Same as in `start_test`. • `tags`: Same as in `start_test`. • `critical`: Same as in `start_test`. • `template`: 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` or `FAIL`. • `message`: Status message. Normally an error message or an empty string.
start_keyword	name, attributes	Called when a keyword starts.

		`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`: String `Keyword` for normal keywords, `Setup` or `Teardown` for the top level keyword used as setup/teardown, `For` for for loops, and `For Item` for individual for loop iterations. **NOTE:** Keyword type reporting was changed in RF 3.0. See issue #2248 for details. • `kwname`: Name of the keyword without library or resource prefix. New in RF 2.9. • `libname`: Name of the library or resource the keyword belongs to, or an empty string when the keyword is in a test case file. New in RF 2.9. • `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. New in RF 2.9. • `tags`: Keyword tags as a list of strings. New in RF 3.0. • `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`. • `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` or `FAIL`.
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 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. Starting from RF 3.0, this method is not called if the message has level below the current threshold level.
message	message	Called when the framework itself writes a syslog 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, `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` with libraries implemented with Java or if getting the source of the library failed for some reason. • `importer`: An absolute path to the file importing the library. `None` when BuiltIn is imported well as when using the *Import Library* keyword. New in Robot Framework 2.9.
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. New in Robot Framework 2.9.
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. New in Robot Framework 2.9.
output_file	path	Called when writing to an output file is ready. `path` is an absolute path to the file.
log_file	path	Called when writing to a log file is ready. `path` is an absolute path to the file.
report_file	path	Called when writing to a report file is ready. `path` is an absolute path to the file.
xunit_file	path	Called when writing to an xunit file is ready. `path` is an absolute path to the file.
debug_file	path	Called when writing to a debug file is ready. `path` is an absolute path to the file.
close		Called when the whole test execution ends. With library listeners called when the library goes out of scope.

The available methods and their arguments are also shown in a formal Java interface specification below. Contents of the `java.util.Map attributes` are as in the table above. It should be remembered that a listener *does not* need to implement any explicit interface or have all these methods.

```
public interface RobotListenerInterface {
    public static final int ROBOT_LISTENER_API_VERSION = 2;
    void startSuite(String name, java.util.Map attributes);
    void endSuite(String name, java.util.Map attributes);
    void startTest(String name, java.util.Map attributes);
    void endTest(String name, java.util.Map attributes);
    void startKeyword(String name, java.util.Map attributes);
    void endKeyword(String name, java.util.Map attributes);
    void logMessage(java.util.Map message);
    void message(java.util.Map message);
    void outputFile(String path);
    void logFile(String path);
    void reportFile(String path);
    void debugFile(String path);
    void close();
}
```

Listener version 3

Listener version 3 has mostly the same methods as 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 was introduced in Robot Framework 3.0. At least initially it does not have all methods that the version 2 has. The main reason is that suitable model objects are not available internally. 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 and its execution results, 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 and its execution results, respectively.
end_test	data, result	Called when a test case ends. Same arguments as with `start_test`.
start_keyword	N/A	Not implemented in RF 3.0.
end_keyword	N/A	Not implemented in RF 3.0.
log_message	message	Called when an executed keyword writes a log message. `message` is a model object representing the logged message. This method is not called if the message has level below the current threshold level.
message	message	Called when the framework itself writes a syslog message. `message` is same object as with `log_message`.
library_import	N/A	Not implemented in RF 3.0.
resource_import	N/A	Not implemented in RF 3.0.
variables_import	N/A	Not implemented in RF 3.0.
output_file	path	Called when writing to an output file is ready. `path` is an absolute path to the file.
log_file	path	Called when writing to a log file is ready. `path` is an absolute path to the file.
report_file	path	Called when writing to a report file is ready. `path` is an absolute path to the file.
xunit_file	path	Called when writing to an xunit file is ready. `path` is an absolute path to the file.
debug_file	path	Called when writing to a debug file is ready. `path` is an absolute path to the file.
close		Called when the whole test execution ends. With library listeners called when the library goes out of scope.

4.3.4 Listeners logging

Robot Framework offers a 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 under the executed keyword.
start_suite, end_suite, start_test, end_test	Messages are logged to the syslog. Warnings are shown also in the execution errors 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

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

The first example is implemented as Python module and uses the 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']))
        raw_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 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))
        raw_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 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()
```

The following example implements the same functionality as the previous one, but uses Java instead of Python.

```
import java.io.*;
import java.util.Map;
import java.util.List;


public class JavaListener {
    public static final int ROBOT_LISTENER_API_VERSION = 2;
    public static final String DEFAULT_FILENAME = "listen_java.txt";
    private BufferedWriter outfile = null;

    public JavaListener() throws IOException {
        this(DEFAULT_FILENAME);
    }

    public JavaListener(String filename) throws IOException {
        String tmpdir = System.getProperty("java.io.tmpdir");
        String sep = System.getProperty("file.separator");
        String outpath = tmpdir + sep + filename;
        outfile = new BufferedWriter(new FileWriter(outpath));
    }

    public void startSuite(String name, Map attrs) throws IOException {
        outfile.write(name + " '" + attrs.get("doc") + "'\n");
    }

    public void startTest(String name, Map attrs) throws IOException {
        outfile.write("- " + name + " '" + attrs.get("doc") + "' [ ");
        List tags = (List)attrs.get("tags");
        for (int i=0; i < tags.size(); i++) {
           outfile.write(tags.get(i) + " ");
        }
        outfile.write(" ] :: ");
    }

    public void endTest(String name, Map attrs) throws IOException {
        String status = attrs.get("status").toString();
```

```
        if (status.equals("PASS")) {
            outfile.write("PASS\n");
        }
        else {
            outfile.write("FAIL: " + attrs.get("message") + "\n");
        }
    }

    public void endSuite(String name, Map attrs) throws IOException {
        outfile.write(attrs.get("status") + "\n" + attrs.get("message") + "\n");
    }

    public void close() throws IOException {
        outfile.close();
    }
}
```

Modifying execution and results

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.

Modifying executed suites and tests

Changing what is executed requires modifying the model object containing the executed test suite or test case 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 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 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 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 and test case result objects passed as the second argument to `start_suite` and `start_test` methods, respectively, and by modifying the message object passed to the `log_message` method. This is demonstrated by the following listener that is implemented as a class.

```
class ResultModifier(object):
    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 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 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 Test libraries as listeners

Sometimes it is useful also for 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

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` or `endSuite`, it is possible to use `_end_suite` or `_endSuite`.

Following examples illustrates using an external listener as well as library acting as a listener itself:

```
import my.project.Listener;

public class JavaLibraryWithExternalListener {
    public static final Listener ROBOT_LIBRARY_LISTENER = new Listener();
    public static final String ROBOT_LIBRARY_SCOPE = "GLOBAL";
    public static final int ROBOT_LISTENER_API_VERSION = 2;

    // actual library code here ...
}
```

```
class PythonLibraryAsListenerItself(object):
    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 using `ROBOT_LISTENER_API_VERSION` attribute exactly like any other listener.

Starting from version 2.9, you can also provide any list like object of instances in the `ROBOT_LIBRARY_LISTENER` attribute. This will cause all instances of the list to be registered as listeners.

Called listener methods

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 test 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 section above for more information about all these methods.

4.4 Extending the Robot Framework Jar

Adding additional test libraries or support code to the Robot Framework jar is quite straightforward using the `jar` command included in standard JDK installation. Python code must be placed in *Lib* directory inside the jar and Java code can be placed directly to the root of the jar, according to package structure.

For example, to add Python package `mytestlib` to the jar, first copy the *mytestlib* directory under a directory called *Lib*, then run following command in the directory containing *Lib*:

```
jar uf /path/to/robotframework-2.7.1.jar Lib
```

To add compiled java classes to the jar, you must have a directory structure corresponding to the Java package structure and add that recursively to the zip.

For example, to add class `MyLib.class`, in package `org.test`, the file must be in *org/test/MyLib.class* and you can execute:

```
jar uf /path/to/robotframework-2.7.1.jar org
```

5 Supporting Tools

5.1 Library documentation tool (Libdoc)

Libdoc is Robot Framework's built-in tool for generating keyword documentation for test libraries and resource files in HTML and XML formats. The former format is suitable for humans and the latter for RIDE and other tools. Libdoc also has few special commands to show library or resource information on the console.

Documentation can be created for:

- test libraries implemented with Python or Java using the normal static library API,
- test libraries using the dynamic API, including remote libraries, and
- resource files.

Additionally it is possible to use XML spec created by Libdoc earlier as an input.

5.1.1 General usage

Synopsis

```
python -m robot.libdoc [options] library_or_resource output_file
python -m robot.libdoc [options] library_or_resource list|show|version [names]
```

Options

`-f, --format <html|xml>`
: Specifies whether to generate HTML or XML output. If this options is not used, the format is got from the extension of the output file.

`-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 got from the source code.

`-P, --pythonpath <path>`
: Additional locations where to search for libraries and resources similarly as when running tests.

`-E, --escape <what:with>`
: Deprecated. Use console escape mechanism instead.

`-h, --help`
: Prints this help.

Alternative execution

Although Libdoc is used only with Python in the synopsis above, it works also with Jython and

IronPython. When documenting Java libraries, Jython is actually required.

In the synopsis Libdoc is executed as an installed module (`python -m robot.libdoc`). In addition to that, it can be run also as a script:

```
python path/robot/libdoc.py [options] arguments
```

Executing as a script can be useful if you have done manual installation or otherwise just have the *robot* directory with the source code somewhere in your system.

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, it is possible to specify the library either by using just the library name or path to the library source code. In the former case the library is searched using the module search path and its name must be in the same format as 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.

Java libraries with path

A Java test library implemented using the static library API can be specified by giving the path to the source code file containing the library implementation. When using Java 9 or newer, documentation can be generated without external dependencies, but with older Java versions the *tools.jar*, which is part of the Java JDK distribution, must be found from the `CLASSPATH` when Libdoc is executed. Notice that generating documentation for Java libraries works only with Jython.

> **Note**
>
> Generating documentation without *tools.jar* when using Java 9 or newer is a new feature in Robot Framework 3.1.

Resource files with path

Resource files must always be specified using a path. If the path does not exist, resource files are also searched from all directories in the module search path similarly as when executing test cases.

Generating documentation

When generating documentation in HTML or XML format, the output file must be specified as the second argument after the library/resource name or path. Output format is got automatically from the extension but can also be set using the `--format` option.

Examples:

```
python -m robot.libdoc OperatingSystem OperatingSystem.html
python -m robot.libdoc --name MyLibrary Remote::http://10.0.0.42:8270 MyLibrary.xml
python -m robot.libdoc test/resource.html doc/resource_doc.html
jython -m robot.libdoc --version 1.0 MyJavaLibrary.java MyJavaLibrary.html
jython -m robot.libdoc my.organization.DynamicJavaLibrary my.organization.DynamicJavaLibrary.xml
```

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:

```
python -m robot.libdoc Dialogs list
python -m robot.libdoc SeleniumLibrary list browser
python -m robot.libdoc Remote::10.0.0.42:8270 show
python -m robot.libdoc Dialogs show PauseExecution execute*
python -m robot.libdoc SeleniumLibrary show intro
python -m robot.libdoc SeleniumLibrary version
```

5.1.2 Writing documentation

This section discusses writing documentation for Python and Java based test libraries that use the static library API as well as for dynamic libraries and resource files. Creating test libraries and 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 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, and there is a bit longer example at the end of this chapter containing also an example of the generated documentation.

```
class ExampleLib:
    """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
```

> **Tip**
>
> If you want to use non-ASCII charactes in the documentation of Python libraries, you must either use UTF-8 as your source code encoding or create docstrings as Unicode.
>
> For more information on Python documentation strings, see PEP-257.

Java libraries

Documentation for Java libraries that use the static library API is written as normal Javadoc comments for the library class and methods. In this case Libdoc actually uses the Javadoc tool internally, and thus *tools.jar* containing it must be in `CLASSPATH`. This jar file is part of the normal Java SDK distribution and ought to be found from *bin* directory under the Java SDK installation.

The following simple example has exactly same documentation (and functionality) than the earlier Python example.

```
/**
 * Library for demo purposes.
 *
 * This library is only used in an example and it doesn't do anything useful.
 */
public class ExampleLib {

    /**
     * Does nothing.
     */
    public void myKeyword() {
    }

    /**
     * Takes one argument and *does nothing* with it.
     *
     * Examples:
     * | Your Keyword | xxx |
     * | Your Keyword | yyy |
```

```
     */
    public void yourKeyword(String arg) {
    }
}
```

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 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. For a Python library, if it has an `__init__` method that takes arguments in addition to `self`, its documentation and arguments are shown. For a Java library, if it has a public constructor that accepts arguments, all its public constructors 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 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 table 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

Libdoc supports documentation in Robot Framework's own documentation syntax, HTML, plain text, and reStructuredText. The format to use can be specified in test library source code 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 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."""
```

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 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 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 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 module installed to be able to generate documentation using it. Because backtick characters have special meaning in reStructuredText, 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. The code block syntax has always worked with Robot Framework, but they are highlighted only in RF 3.0.1 and newer. Syntax highlight requires additional Pygments 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

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 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 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, backticks must be escaped like `` \`Keyword Name\` ``.

Linking to automatic sections

The documentation generated by Libdoc always contains sections for overall library introduction, shortcuts to keywords, and for actual keywords. If a library itself takes arguments, there is also separate importing section.

All these 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` ``
Shortcuts	`` `shortcuts` ``
Keywords	`` `keywords` ``

Linking to custom sections

Robot Framework's documentation syntax supports custom 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.

5.1.5 Representing arguments

Libdoc handles keywords' arguments automatically so that arguments specified for methods in libraries or user keywords in resource files are listed in a separate column. User keyword arguments are shown without `${}` or `@{}` to make arguments look the same regardless where keywords originated from.

Regardless how keywords are actually implemented, Libdoc shows arguments similarly as when creating keywords in Python. This formatting is explained more thoroughly in the table below.

How Libdoc represents arguments

Arguments	Now represented	Examples
No arguments	Empty column.	
One or more argument	List of strings containing argument names.	`one_argument` `a1, a2, a3`
Default values for arguments	Default values separated from names with `=`.	`arg=default value` `a, b=1, c=2`
Variable number of arguments (varargs)	Last (or second last with kwargs) argument has `*` before its name.	`*varargs` `a, b=42, *rest`

Free keyword arguments (kwargs)	Last arguments has `**` before its name.	`**kwargs`
		`a, b=42, **kws`
		`*varargs, **kwargs`

When referring to arguments in keyword documentation, it is recommended to use inline code style like ``` ``argument`` ```.

5.1.6 Libdoc example

The following example illustrates how to use the most important documentation formatting possibilities, internal linking, and so on. Click here to see how the generated documentation looks like.

```
class LoggingLibrary:
    """Library for logging messages.

    = Table of contents =

    - `Usage`
    - `Valid log levels`
    - `Examples`
    - `Importing`
    - `Shortcuts`
    - `Keywords`

    = 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 have documentation generated by Libdoc and their documentation (and source code) act as a more realistic examples.

5.2 Test data documentation tool (Testdoc)

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

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 when running tests. New in Robot Framework 3.0.2.

`-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.

5.2.2 Generating documentation

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 works with all interpreters supported by Robot Framework (Python, Jython and IronPython). It 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
jython -m robot.testdoc --name "Smoke tests" --include smoke path/to/tests smoke.html
ipy path/to/robot/testdoc.py first.robot second.robot output.html
```

5.3 Test data clean-up tool (Tidy)

Tidy is Robot Framework's built-in a tool for cleaning up and changing the format of Robot Framework test data files.

The output is written into the standard output stream by default, but an optional output file can be given as well. Files can also be modified in-place using `--inplace` or `--recursive` options.

5.3.1 General usage

Synopsis

```
python -m robot.tidy [options] inputfile
python -m robot.tidy [options] inputfile [outputfile]
python -m robot.tidy --inplace [options] inputfile [more input files]
python -m robot.tidy --recursive [options] directory
```

Options

`-i, --inplace`
: Tidy given file(s) so that original file(s) are overwritten (or removed, if the format is changed). When this option is used, it is possible to give multiple input files. Examples:

```
python -m robot.tidy --inplace tests.robot
python -m robot.tidy --inplace --format robot *.html
```

`-r, --recursive`
: Process given directory recursively. Files in the directory are processed in place similarly as when `--inplace` option is used.

`-f, --format <robot|txt|html|tsv>`
: Output file format. If the output file is given explicitly, the default value is got from its extension. Otherwise the format is not changed.

`-p, --use-pipes`
: Use a pipe character (|) as a cell separator in the plain text format.

`-s, --spacecount <number>`
: The number of spaces between cells in the plain text format. Default is 4.

`-l, --lineseparator <native|windows|unix>`
: Line separator to use in outputs. The default is 'native'.

- *native*: use operating system's native line separators
- *windows*: use Windows line separators (CRLF)

- *unix*: use Unix line separators (LF)

`-h, --help` Show this help.

Alternative execution

Although Tidy is used only with Python in the synopsis above, it works also with Jython and IronPython. In the synopsis Tidy is executed as an installed module (`python -m robot.tidy`), but it can be run also as a script:

```
python path/robot/tidy.py [options] arguments
```

Executing as a script can be useful if you have done manual installation or otherwise just have the *robot* directory with the source code somewhere in your system.

Output encoding

All output files are written using UTF-8 encoding. Outputs written to the console use the current console encoding.

5.3.2 Cleaning up test data

Test case files can be normalized using Tidy. Tidy always writes consistent headers, consistent order for settings, and consistent amount of whitespace between sections and cells.

Examples:

```
python -m robot.tidy messed_up_tests.robot cleaned_up_tests.robot
python -m robot.tidy --inplace tests.robot
python -m robot.tidy --recursive path/to/tests
```

5.3.3 Changing test data format

Robot Framework supports test data in various formats, but nowadays the plain text format with the *.robot* extension is the most commonly used. Tidy makes it easy to convert data from one format to another. This is especially useful if there is a need to convert tests in deprecated HTML format to other formats.

Input format is always determined based on the extension of the input file. If output file is given, the output format is got from its extension. When using `--inplace` or `--recursive`, it is possible to specify the desired format using the `--format` option.

Examples:

```
python -m robot.tidy tests.html tests.robot
python -m robot.tidy --format robot --inplace tests.html
python -m robot.tidy --format robot --recursive path/to/tests
```

5.4 External tools

There are plenty of external tools that can be used with Robot Framework. These tools include test data editor RIDE, extensions for various IDEs and text editors, plugins to continuous integration systems and build tools, 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/#tools.

6 Appendices

6.1 All available settings in test data

6.1.1 Setting table

The Setting table 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 table can only include settings for importing libraries, resources, and variables.

Settings available in the Setting table

Name	Description
Library	Used for importing libraries.
Resource	Used for taking resource files into use.
Variables	Used for taking variable files into use.
Documentation	Used for specifying a test suite or resource file documentation.
Metadata	Used for setting free test suite metadata.
Suite Setup	Used for specifying the suite setup.
Suite Teardown	Used for specifying the suite teardown.
Force Tags	Used for specifying forced values for tags when tagging test cases.
Default Tags	Used for specifying default values for tags when tagging test cases.
Test Setup	Used for specifying a default test setup.
Test Teardown	Used for specifying a default test teardown.
Test Template	Used for specifying a default template keyword for test cases.
Test Timeout	Used for specifying a default 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.

Note

All setting names can optionally include a colon at the end, for example *Documentation:*. This can make reading the settings easier especially when using the plain text format.

6.1.2 Test Case table

The settings in the Test Case table are always specific to the test case for which they are defined. Some of these settings override the default values defined in the Settings table.

Exactly same settings are available when creating tasks in the Task table.

Settings available in the Test Case table

Name	Description
[Documentation]	Used for specifying a test case documentation.
[Tags]	Used for tagging test cases.
[Setup]	Used for specifying a test setup.
[Teardown]	Used for specifying a test teardown.
[Template]	Used for specifying a template keyword.
[Timeout]	Used for specifying a test case timeout.

6.1.3 Keyword table

Settings in the Keyword table are specific to the user keyword for which they are defined.

Settings available in the Keyword table

Name	Description
[Documentation]	Used for specifying a user keyword documentation.
[Tags]	Used for specifying user keyword tags.
[Arguments]	Used for specifying user keyword arguments.
[Return]	Used for specifying user keyword return values.
[Teardown]	Used for specifying user keyword teardown.
[Timeout]	Used for specifying a user keyword timeout.

6.2 All command line options

This appendix lists all the command line options that are available when executing test cases and when post-processing outputs. Also environment variables affecting execution and post-processing are listed.

6.2.3 Environment variables for execution and post-processing

6.2.1 Command line options for test execution

`--rpa` Turn on generic automation mode.

`-F, --extension <value>`
Parse only these files when executing a directory.

`-N, --name <name>`
Sets the name of the top-level test suite.

`-D, --doc <document>`
Sets the documentation of the top-level test suite.

`-M, --metadata <name:value>`
Sets free metadata for the top level test suite.

`-G, --settag <tag>`
Sets the tag(s) to all executed test cases.

`-t, --test <name>`
Selects the test cases by name.

`--task <name>` Alias for `--test` that can be used when executing tasks.

`-s, --suite <name>`
Selects the test suites by name.

`-R, --rerunfailed <file>`
Selects failed tests from an earlier output file to be re-executed.

`-S, --rerunfailedsuites <file>`
Selects failed test suites from an earlier output file to be re-executed.

`-i, --include <tag>`
Selects the test cases by tag.

`-e, --exclude <tag>`
Selects the test cases by tag.

`-c, --critical <tag>`
Tests that have the given tag are considered critical.

`-n, --noncritical <tag>`
Tests that have the given tag are not critical.

`-v, --variable <name:value>`
Sets individual variables.

`-V, --variablefile <path:args>`
Sets variables using variable files.

`-d, --outputdir <dir>`
Defines where to create output files.

`-o, --output <file>`
Sets the path to the generated output file.

`-l, --log <file>`
Sets the path to the generated log file.

`-r, --report <file>`
Sets the path to the generated report file.

`-x, --xunit <file>`
Sets the path to the generated xUnit compatible result file.

`--xunitskipnoncritical`
Mark non-critical tests on xUnit compatible result file as skipped.

`-b, --debugfile <file>`
A debug file that is written during execution.

`-T, --timestampoutputs`
Adds a timestamp to all output files.

`--splitlog` Split log file into smaller pieces that open in browser transparently.

`--logtitle <title>`
Sets a title for the generated test log.

`--reporttitle <title>`
Sets a title for the generated test report.

`--reportbackground <colors>`
Sets background colors of the generated report.

`--maxerrorlines <lines>`
Sets the number of error lines shown in reports when tests fail.

`-L, --loglevel <level>`
Sets the threshold level for logging. Optionally the default visible log level can be given separated with a colon (:).

`--suitestatlevel <level>`
Defines how many levels to show in the *Statistics by Suite* table in outputs.

`--tagstatinclude <tag>`
Includes only these tags in the *Statistics by Tag* table.

`--tagstatexclude <tag>`
Excludes these tags from the *Statistics by Tag* table.

`--tagstatcombine <tags:title>`
Creates combined statistics based on tags.

`--tagdoc <pattern:doc>`
Adds documentation to the specified tags.

`--tagstatlink <pattern:link:title>`
: Adds external links to the *Statistics by Tag* table.

`--removekeywords <all|passed|name:pattern|tag:pattern|for|wuks>`
: Removes keyword data from the generated log file.

`--flattenkeywords <for|foritem|name:pattern|tag:pattern>`
: Flattens keywords in the generated log file.

`--listener <name:args>`
: Sets a listener for monitoring test execution.

`--warnonskippedfiles`
: Deprecated. Nowadays all skipped files are reported.

`--nostatusrc`
: Sets the return code to zero regardless of failures in test cases. Error codes are returned normally.

`--runemptysuite`
: Executes tests also if the selected test suites are empty.

`--dryrun`
: In the dry run mode tests are run without executing keywords originating from test libraries. Useful for validating test data syntax.

`-X, --exitonfailure`
: Stops test execution if any critical test fails.

`--exitonerror`
: Stops test execution if any error occurs when parsing test data, importing libraries, and so on.

`--skipteardownonexit`
: Skips teardowns is test execution is prematurely stopped.

`--prerunmodifier <name:args>`
: Activate programmatic modification of test data.

`--prerebotmodifier <name:args>`
: Activate programmatic modification of results.

`--randomize <all|suites|tests|none>`
: Randomizes test execution order.

`--console <verbose|dotted|quiet|none>`
: Console output type.

`--dotted`
: Shortcut for `--console dotted`.

`--quiet`
: Shortcut for `--console quiet`.

`-W, --consolewidth <width>`
: Sets the width of the console output.

`-C, --consolecolors <auto|on|ansi|off>`
: Specifies are colors used on the console.

`-K, --consolemarkers <auto|on|off>`
: Show markers on the console when top level keywords in a test case end.

`-P, --pythonpath <path>`
: Additional locations to add to the module search path.

`-E, --escape <what:with>`
: Deprecated. Use console escape mechanism instead.

`-A, --argumentfile <path>`
: A text file to read more arguments from.

`-h, --help`
: Prints usage instructions.

`--version`
: Prints the version information.

6.2.2 Command line options for post-processing outputs

`--rpa`
: Turn on generic automation mode.

`-R, --merge`
: Changes result combining behavior to merging.

`-N, --name <name>`
: Sets the name of the top level test suite.

`-D, --doc <document>`
: Sets the documentation of the top-level test suite.

`-M, --metadata <name:value>`
: Sets free metadata for the top-level test suite.

`-G, --settag <tag>`
: Sets the tag(s) to all processed test cases.

`-t, --test <name>`
: Selects the test cases by name.

`--task <name>`
: Alias for `--test`.

`-s, --suite <name>`
: Selects the test suites by name.

`-i, --include <tag>`
: Selects the test cases by tag.

`-e, --exclude <tag>`
: Selects the test cases by tag.

`-c, --critical <tag>`
: Tests that have the given tag are considered critical.

`-n, --noncritical <tag>`
: Tests that have the given tag are not critical.

`-d, --outputdir <dir>`
: Defines where to create output files.

`-o, --output <file>`
: Sets the path to the generated output file.

`-l, --log <file>`
: Sets the path to the generated log file.

`-r, --report <file>`
: Sets the path to the generated report file.

`-x, --xunit <file>`
: Sets the path to the generated xUnit compatible result file.

`--xunitskipnoncritical`
: Mark non-critical tests on xUnit compatible result file as skipped.

`-T, --timestampoutputs`
: Adds a timestamp to all output files.

`--splitlog`
: Split log file into smaller pieces that open in browser transparently.

`--logtitle <title>`
: Sets a title for the generated test log.

`--reporttitle <title>`
: Sets a title for the generated test report.

`--reportbackground <colors>`
: Sets background colors of the generated report.

`-L, --loglevel <level>`
: Sets the threshold level to select log messages. Optionally the default visible log level can be given separated with a colon (:).

`--suitestatlevel <level>`
: Defines how many levels to show in the *Statistics by Suite* table in outputs.

`--tagstatinclude <tag>`
: Includes only these tags in the *Statistics by Tag* table.

`--tagstatexclude <tag>`
: Excludes these tags from the *Statistics by Tag* table.

`--tagstatcombine <tags:title>`
: Creates combined statistics based on tags.

`--tagdoc <pattern:doc>`
: Adds documentation to the specified tags.

`--tagstatlink <pattern:link:title>`
: Adds external links to the *Statistics by Tag* table.

`--removekeywords <all|passed|name:pattern|tag:pattern|for|wuks>`
: Removes keyword data from the generated outputs.

`--flattenkeywords <for|foritem|name:pattern|tag:pattern>`
: Flattens keywords in the generated outputs.

`--starttime <timestamp>`
: Sets the starting time of test execution when creating reports.

`--endtime <timestamp>`
: Sets the ending time of test execution when creating reports.

`--nostatusrc`
: Sets the return code to zero regardless of failures in test cases. Error codes are returned normally.

`--processemptysuite`
: Processes output files even if files contain empty test suites.

`--prerebotmodifier <name:args>`
: Activate programmatic modification of results.

`-C, --consolecolors <auto|on|ansi|off>`
: Specifies are colors used on the console.

`-P, --pythonpath <path>`
: Additional locations to add to the module search path.

`-E, --escape <what:with>`
: Deprecated. Use console escape mechanism instead.

`-A, --argumentfile <path>`
: A text file to read more arguments from.

`-h, --help`
: Prints usage instructions.

`--version`
: Prints the version information.

6.2.3 Environment variables for execution and post-processing

`ROBOT_OPTIONS` and `REBOT_OPTIONS`
: Space separated list of default options to be placed in front of any explicit options on the command line.

`ROBOT_SYSLOG_FILE`
: Path to a syslog 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 file.

`ROBOT_INTERNAL_TRACES`
: When set to any non-empty value, Robot Framework's internal methods are included in error tracebacks.

6.3 Documentation formatting

It is possible to use simple HTML formatting with test suite, test case and user keyword documentation and free test suite metadata in the test data, as well as when documenting test 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

Newlines in test data

When documenting test suites, test cases and keywords or adding metadata to test suites, newlines can be added manually using `\n` 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 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 constructing lists, 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. 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. If documentation or metadata is defined in multiple columns, cells in a same row are concatenated together with spaces. This kind of splitting can be a good idea especially when using the HTML format and columns are narrow. 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

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

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

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, 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

All strings that look like URLs are automatically converted into clickable links. Additionally, URLs that end with extension *.jpg*, *.jpeg*, *.png*, *.gif* or *.bmp* (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.

6.3.5 Custom links and images

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. The surrounding square brackets and the pipe character between the parts are mandatory in all cases.

Link with text content

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

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>
[image.jpg|thumb.jpg] -> <a href="image.jpg"><img src="thumb.jpg"></a>
```

Image with title text

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!">
```

6.3.6 Section titles

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

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

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

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 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

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

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 keywords *Sleep* and *Wait Until Keyword Succeeds*), DateTime library, and timeouts.

6.4.1 Time as number

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.

6.4.2 Time as time string

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

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

Many keywords in Robot Framework 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 either empty or case-insensitively equal to `false` or `no`. Other strings are considered true regardless their value, and other argument types are tested using same rules as in Python.

Keyword can also accept other special strings than `false` and `no` that are to be considered false. For example, BuiltIn keyword `Should Be True` used in the examples below considers string `no values` given to its `values` argument as false.

```
*** 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 in this context.
```

Note that prior to Robot Framework 2.9 handling Boolean arguments was inconsistent. Some keywords followed the above rules, but others simply considered all non-empty strings, including `false` and `no`, to be true.

6.6 Internal API

API documentation is hosted separately at the excellent Read the Docs service. If you are unsure how to use certain API or is using them forward compatible, please send a question to mailing list.
