Robot Framework Documentation

Release 4.0.3

Robot Framework developers

May 25, 2021

Contents

This documentation describes the public API of Robot Framework. Installation, basic usage and wealth of other topics are covered by the Robot Framework User Guide.

Main API entry points are documented here, but the lower level implementation details are not always that well documented. If the documentation is insufficient, it is possible to view the source code by clicking `[source]` link in the documentation. In case viewing the source is not helpful either, questions may be sent to the robotframework-users mailing list.

CHAPTER 1

Entry points

Command line entry points are implemented as Python modules and they also provide programmatic APIs. Following entry points exist:

- `robot.run` entry point for executing tests.
- `robot.rebot` entry point for post-processing outputs (Rebot).
- `robot.libdoc` entry point for Libdoc tool.
- `robot.testdoc` entry point for Testdoc tool.
- `robot.tidy` entry point for Tidy tool.

See built-in tool documentation for more details about Rebot, Libdoc, Testdoc, and Tidy tools.

CHAPTER 2

Java entry points

The Robot Framework Jar distribution contains also a Java API, in the form of the org.robotframework.RobotFramework class.

CHAPTER 3

Public API

robot.api package exposes the public APIs of Robot Framework.

Unless stated otherwise, the APIs exposed in this package are considered stable, and thus safe to use when building external tools on top of Robot Framework. Notice that all parsing APIs were rewritten in Robot Framework 3.2.

Currently exposed APIs are:

- *logger* module for libraries' logging purposes.
- *deco* module with decorators libraries can utilize.
- *exceptions* module containing exceptions that libraries can utilize for reporting failures and other events. These exceptions can be imported also directly via *robot.api* like `from robot.api import SkipExecution`.
- *parsing* module exposing the parsing APIs. This module is new in Robot Framework 4.0. Various parsing related functions and classes were exposed directly via *robot.api* already in Robot Framework 3.2, but they are effectively deprecated and will be removed in the future.
- *TestSuite* class for creating executable test suites programmatically and *TestSuiteBuilder* class for creating such suites based on existing test data on the file system.
- *SuiteVisitor* abstract class for processing testdata before execution. This can be used as a base for implementing a pre-run modifier that is taken into use with `--prerunmodifier` commandline option.
- *ExecutionResult()* factory method for reading execution results from XML output files and *ResultVisitor* abstract class to ease further processing the results. *ResultVisitor* can also be used as a base for pre-Rebot modifier that is taken into use with `--prerebotmodifier` commandline option.
- *ResultWriter* class for writing reports, logs, XML outputs, and XUnit files. Can write results based on XML outputs on the file system, as well as based on the result objects returned by the *ExecutionResult()* or an executed *TestSuite*.

All of the above names can be imported like:

```
from robot.api import ApiName
```

See documentations of the individual APIs for more details.

Tip: APIs related to the command line entry points are exposed directly via the `robot` root package.

CHAPTER 4

All packages

All *robot* packages are listed below. Typically you should not need to import anything from them directly, but the above public APIs may return objects implemented in them.

4.1 robot package

The root of the Robot Framework package.

The command line entry points provided by the framework are exposed for programmatic usage as follows:

- *run()*: Function to run tests.
- *run_cli()*: Function to run tests with command line argument processing.
- *rebot()*: Function to post-process outputs.
- *rebot_cli()*: Function to post-process outputs with command line argument processing.
- *libdoc*: Module for library documentation generation.
- *testdoc*: Module for test case documentation generation.
- *tidy*: Module for test data clean-up and format change.

All the functions above can be imported like `from robot import run`. Functions and classes provided by the modules need to be imported like `from robot.libdoc import libdoc_cli`.

The functions and modules listed above are considered stable. Other modules in this package are for for internal usage and may change without prior notice.

Tip: More public APIs are exposed by the *robot.api* package.

robot.**run**(**tests*, ***options*)

Programmatic entry point for running tests.

Parameters

- **tests** – Paths to test case files/directories to be executed similarly as when running the `robot` command on the command line.
- **options** – Options to configure and control execution. Accepted options are mostly same as normal command line options to the `robot` command. Option names match command line option long names without hyphens so that, for example, `--name` becomes `name`.

Most options that can be given from the command line work. An exception is that options `--pythonpath`, `--argumentfile`, `--help` and `--version` are not supported.

Options that can be given on the command line multiple times can be passed as lists. For example, `include=['tag1', 'tag2']` is equivalent to `--include tag1 --include tag2`. If such options are used only once, they can be given also as a single string like `include='tag'`.

Options that accept no value can be given as Booleans. For example, `dryrun=True` is same as using the `--dryrun` option.

Options that accept string `NONE` as a special value can also be used with Python `None`. For example, using `log=None` is equivalent to `--log NONE`.

`listener`, `prerunmodifier` and `prerebotmodifier` options allow passing values as Python objects in addition to module names these command line options support. For example, `run('tests', listener=MyListener())`.

To capture the standard output and error streams, pass an open file or file-like object as special keyword arguments `stdout` and `stderr`, respectively.

A return code is returned similarly as when running on the command line. Zero means that tests were executed and no critical test failed, values up to 250 denote the number of failed critical tests, and values between 251-255 are for other statuses documented in the Robot Framework User Guide.

Example:

```
from robot import run

run('path/to/tests.robot')
run('tests.robot', include=['tag1', 'tag2'], splitlog=True)
with open('stdout.txt', 'w') as stdout:
    run('t1.robot', 't2.robot', name='Example', log=None, stdout=stdout)
```

Equivalent command line usage:

```
robot path/to/tests.robot
robot --include tag1 --include tag2 --splitlog tests.robot
robot --name Example --log NONE t1.robot t2.robot > stdout.txt
```

robot.**run_cli**(*arguments=None*, *exit=True*)

Command line execution entry point for running tests.

Parameters

- **arguments** – Command line options and arguments as a list of strings. Starting from RF 3.1, defaults to `sys.argv[1:]` if not given.
- **exit** – If `True`, call `sys.exit` with the return code denoting execution status, otherwise just return the rc.

Entry point used when running tests from the command line, but can also be used by custom scripts that execute tests. Especially useful if the script itself needs to accept same arguments as accepted by Robot Framework, because the script can just pass them forward directly along with the possible default values it sets itself.

Example:

```
from robot import run_cli

# Run tests and return the return code.
rc = run_cli(['--name', 'Example', 'tests.robot'], exit=False)

# Run tests and exit to the system automatically.
run_cli(['--name', 'Example', 'tests.robot'])
```

See also the *run()* function that allows setting options as keyword arguments like `name="Example"` and generally has a richer API for programmatic test execution.

`robot.`**`rebot`**(**outputs*, ***options*)

Programmatic entry point for post-processing outputs.

Parameters

- **`outputs`** – Paths to Robot Framework output files similarly as when running the `rebot` command on the command line.
- **`options`** – Options to configure processing outputs. Accepted options are mostly same as normal command line options to the `rebot` command. Option names match command line option long names without hyphens so that, for example, `--name` becomes `name`.

The semantics related to passing options are exactly the same as with the *run()* function. See its documentation for more details.

Examples:

```
from robot import rebot

rebot('path/to/output.xml')
with open('stdout.txt', 'w') as stdout:
    rebot('o1.xml', 'o2.xml', name='Example', log=None, stdout=stdout)
```

Equivalent command line usage:

```
rebot path/to/output.xml
rebot --name Example --log NONE o1.xml o2.xml > stdout.txt
```

`robot.`**`rebot_cli`**(*arguments=None*, *exit=True*)

Command line execution entry point for post-processing outputs.

Parameters

- **`arguments`** – Command line options and arguments as a list of strings. Starting from RF 3.1, defaults to `sys.argv[1:]` if not given.
- **`exit`** – If `True`, call `sys.exit` with the return code denoting execution status, otherwise just return the rc.

Entry point used when post-processing outputs from the command line, but can also be used by custom scripts. Especially useful if the script itself needs to accept same arguments as accepted by Rebot, because the script can just pass them forward directly along with the possible default values it sets itself.

Example:

```
from robot import rebot_cli

rebot_cli(['--name', 'Example', '--log', 'NONE', 'o1.xml', 'o2.xml'])
```

See also the *rebot()* function that allows setting options as keyword arguments like `name="Example"` and generally has a richer API for programmatic Rebot execution.

4.1.1 Subpackages

robot.api package

robot.api package exposes the public APIs of Robot Framework.

Unless stated otherwise, the APIs exposed in this package are considered stable, and thus safe to use when building external tools on top of Robot Framework. Notice that all parsing APIs were rewritten in Robot Framework 3.2.

Currently exposed APIs are:

- *logger* module for libraries' logging purposes.
- *deco* module with decorators libraries can utilize.
- *exceptions* module containing exceptions that libraries can utilize for reporting failures and other events. These exceptions can be imported also directly via *robot.api* like `from robot.api import SkipExecution`.
- *parsing* module exposing the parsing APIs. This module is new in Robot Framework 4.0. Various parsing related functions and classes were exposed directly via *robot.api* already in Robot Framework 3.2, but they are effectively deprecated and will be removed in the future.
- *TestSuite* class for creating executable test suites programmatically and *TestSuiteBuilder* class for creating such suites based on existing test data on the file system.
- *SuiteVisitor* abstract class for processing testdata before execution. This can be used as a base for implementing a pre-run modifier that is taken into use with `--prerunmodifier` commandline option.
- *ExecutionResult()* factory method for reading execution results from XML output files and *ResultVisitor* abstract class to ease further processing the results. *ResultVisitor* can also be used as a base for pre-Rebot modifier that is taken into use with `--prerebotmodifier` commandline option.
- *ResultWriter* class for writing reports, logs, XML outputs, and XUnit files. Can write results based on XML outputs on the file system, as well as based on the result objects returned by the *ExecutionResult()* or an executed *TestSuite*.

All of the above names can be imported like:

```
from robot.api import ApiName
```

See documentations of the individual APIs for more details.

Tip: APIs related to the command line entry points are exposed directly via the *robot* root package.

Submodules

robot.api.deco module

robot.api.deco.**not_keyword**(*func*)

Decorator to disable exposing functions or methods as keywords.

Examples:

```
@not_keyword
def not_exposed_as_keyword():
    # ...

def exposed_as_keyword():
    # ...
```

Alternatively the automatic keyword discovery can be disabled with the *library()* decorator or by setting the ROBOT_AUTO_KEYWORDS attribute to a false value.

New in Robot Framework 3.2.

robot.api.deco.**keyword**(*name=None*, *tags=()*, *types=()*)

Decorator to set custom name, tags and argument types to keywords.

This decorator creates robot_name, robot_tags and robot_types attributes on the decorated keyword function or method based on the provided arguments. Robot Framework checks them to determine the keyword's name, tags, and argument types, respectively.

Name must be given as a string, tags as a list of strings, and types either as a dictionary mapping argument names to types or as a list of types mapped to arguments based on position. It is OK to specify types only to some arguments, and setting types to None disables type conversion altogether.

If the automatic keyword discovery has been disabled with the *library()* decorator or by setting the ROBOT_AUTO_KEYWORDS attribute to a false value, this decorator is needed to mark functions or methods keywords.

Examples:

```
@keyword
def example():
    # ...

@keyword('Login as user "${user}" with password "${password}"',
         tags=['custom name', 'embedded arguments', 'tags'])
def login(user, password):
    # ...

@keyword(types={'length': int, 'case_insensitive': bool})
def types_as_dict(length, case_insensitive):
    # ...

@keyword(types=[int, bool])
def types_as_list(length, case_insensitive):
    # ...

@keyword(types=None)
def no_conversion(length, case_insensitive=False):
    # ...
```

robot.api.deco.**library**(*scope=None*, *version=None*, *doc_format=None*, *listener=None*, *auto_keywords=False*)

Class decorator to control keyword discovery and other library settings.

By default disables automatic keyword detection by setting class attribute ROBOT_AUTO_KEYWORDS = False to the decorated library. In that mode only methods decorated explicitly with the *keyword()* decorator become keywords. If that is not desired, automatic keyword discovery can be enabled by using auto_keywords=True.

Arguments `scope`, `version`, `doc_format` and `listener` set the library scope, version, documentation format and listener by using class attributes `ROBOT_LIBRARY_SCOPE`, `ROBOT_LIBRARY_VERSION`, `ROBOT_LIBRARY_DOC_FORMAT` and `ROBOT_LIBRARY_LISTENER`, respectively. These attributes are only set if the related arguments are given and they override possible existing attributes in the decorated class.

Examples:

```
@library
class KeywordDiscovery:

    @keyword
    def do_something(self):
        # ...

    def not_keyword(self):
        # ...


@library(scope='GLOBAL', version='3.2')
class LibraryConfiguration:
    # ...
```

The `@library` decorator is new in Robot Framework 3.2.

robot.api.exceptions module

Exceptions that libraries can use for communicating failures and other events.

These exceptions can be imported also via the top level *robot.api* package like `from robot.api import SkipExecution`.

This module and all exceptions are new in Robot Framework 4.0.

exception robot.api.exceptions.**Failure**(*message*, *html=False*)

Bases: exceptions.AssertionError

Report failed validation.

There is no practical difference in using this exception compared to using the standard `AssertionError`. The main benefits are HTML support and that the name of this exception is consistent with other exceptions in this module.

Parameters

- **message** – Exception message.
- **html** – When `True`, message is considered to be HTML and not escaped.

ROBOT_SUPPRESS_NAME = True

args

message

exception robot.api.exceptions.**ContinuableFailure**(*message*, *html=False*)

Bases: *robot.api.exceptions.Failure*

Report failed validation but allow continuing execution.

Parameters

- **message** – Exception message.

- **html** – When `True`, message is considered to be HTML and not escaped.

ROBOT_CONTINUE_ON_FAILURE = True

ROBOT_SUPPRESS_NAME = True

args

message

exception robot.api.exceptions.**Error**(*message*, *html=False*)
Bases: exceptions.RuntimeError

Report error in execution.

Failures related to the system not behaving as expected should typically be reported using the *Failure* exception or the standard `AssertionError`. This exception can be used, for example, if the keyword is used incorrectly.

There is no practical difference in using this exception compared to using the standard `RuntimeError`. The main benefits are HTML support and that the name of this exception is consistent with other exceptions in this module.

Parameters

- **message** – Exception message.
- **html** – When `True`, message is considered to be HTML and not escaped.

ROBOT_SUPPRESS_NAME = True

args

message

exception robot.api.exceptions.**FatalError**(*message*, *html=False*)
Bases: *robot.api.exceptions.Error*

Report error that stops the whole execution.

Parameters

- **message** – Exception message.
- **html** – When `True`, message is considered to be HTML and not escaped.

ROBOT_EXIT_ON_FAILURE = True

ROBOT_SUPPRESS_NAME = False

args

message

exception robot.api.exceptions.**SkipExecution**(*message*, *html=False*)
Bases: exceptions.Exception

Mark the executed test or task skipped.

Parameters

- **message** – Exception message.
- **html** – When `True`, message is considered to be HTML and not escaped.

ROBOT_SKIP_EXECUTION = True

ROBOT_SUPPRESS_NAME = True

args

message

robot.api.logger module

Public logging API for test libraries.

This module provides a public API for writing messages to the log file and 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 cleaner to use, this API has a benefit that the log messages have accurate timestamps.

If the logging methods are used when Robot Framework is not running, the messages are redirected to the standard Python `logging` module using logger named `RobotFramework`.

Log levels

It is possible to log messages using levels `TRACE`, `DEBUG`, `INFO`, `WARN` and `ERROR` either using the *`write()`* function or, more commonly, with the log level specific *`trace()`*, *`debug()`*, *`info()`*, *`warn()`*, *`error()`* functions.

By default the trace and debug messages are not logged but that can be changed with the `--loglevel` command line option. Warnings and errors are automatically written also to the console and to the *Test Execution Errors* section in the log file.

Logging HTML

All methods that are used for writing messages to the log file have an optional `html` argument. If a message to be logged is supposed to be shown as HTML, this argument should be set to `True`. Alternatively, *`write()`* accepts a pseudo log level `HTML`.

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)
```

robot.api.logger.**write**(*msg*, *level='INFO'*, *html=False*)

Writes the message to the log file using the given level.

Valid log levels are `TRACE`, `DEBUG`, `INFO` (default), `WARN`, and `ERROR`. Additionally it is possible to use `HTML` pseudo log level that logs the message as HTML using the `INFO` level.

Instead of using this method, it is generally better to use the level specific methods such as `info` and `debug` that have separate `html` argument to control the message format.

`robot.api.logger.`**`trace`**(*msg*, *html=False*)
Writes the message to the log file using the `TRACE` level.

`robot.api.logger.`**`debug`**(*msg*, *html=False*)
Writes the message to the log file using the `DEBUG` level.

`robot.api.logger.`**`info`**(*msg*, *html=False*, *also_console=False*)
Writes the message to the log file using the `INFO` level.

If `also_console` argument is set to `True`, the message is written both to the log file and to the console.

`robot.api.logger.`**`warn`**(*msg*, *html=False*)
Writes the message to the log file using the `WARN` level.

`robot.api.logger.`**`error`**(*msg*, *html=False*)
Writes the message to the log file using the `ERROR` level.

`robot.api.logger.`**`console`**(*msg*, *newline=True*, *stream='stdout'*)
Writes the message to the console.

If the `newline` argument is `True`, a newline character is automatically added to the message.

By default the message is written to the standard output stream. Using the standard error stream is possibly by giving the `stream` argument value `'stderr'`.

robot.api.parsing module

Public API for parsing, inspecting and modifying test data.

Exposed API

The publicly exposed parsing entry points are the following:

- *`get_tokens()`*, *`get_resource_tokens()`*, and *`get_init_tokens()`* functions for *parsing data to tokens*.
- *`Token`* class that contains all token types as class attributes.
- *`get_model()`*, *`get_resource_model()`*, and *`get_init_model()`* functions for *parsing data to model* represented as an abstract syntax tree (AST).
- *Model objects* used by the AST model.
- *`ModelVisitor`* to ease *inspecting model* and *modifying data*.
- *`ModelTransformer`* for *adding and removing nodes*.

Note: This module is new in Robot Framework 4.0. In Robot Framework 3.2 functions for getting tokens and model as well as the *`Token`* class were exposed directly via the *`robot.api`* package, but other parts of the parsing API were not publicly exposed. All code targeting Robot Framework 4.0 or newer should use this module because parsing related functions and classes will be removed from *`robot.api`* in the future.

Note: Parsing was totally rewritten in Robot Framework 3.2 and external tools using the parsing APIs need to be updated. Depending on the use case, it may be possible to use the higher level *`TestSuiteBuilder()`* instead.

Parsing data to tokens

Data can be parsed to tokens by using *get_tokens()*, *get_resource_tokens()* or *get_init_tokens()* functions depending on whether the data represent a test case (or task) file, a resource file, or a suite initialization file. In practice the difference between these functions is what settings and sections are valid.

Typically the data is easier to inspect and modify by using the higher level model discussed in the next section, but in some cases having just the tokens can be enough. Tokens returned by the aforementioned functions are *Token* instances and they have the token type, value, and position easily available as their attributes. Tokens also have useful string representation used by the example below:

```
from robot.api.parsing import get_tokens

path = 'example.robot'

for token in get_tokens(path):
    print(repr(token))
```

If the `example.robot` used by the above example would contain

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

Second example
    Keyword    xxx

*** Keywords ***
Keyword
    [Arguments]    ${arg}
    Log    ${arg}
```

then the beginning of the output got when running the earlier code would look like this:

```
Token(TESTCASE_HEADER, '*** Test Cases ***', 1, 0)
Token(EOL, '\n', 1, 18)
Token(EOS, '', 1, 19)
Token(TESTCASE_NAME, 'Example', 2, 0)
Token(EOL, '\n', 2, 7)
Token(EOS, '', 2, 8)
Token(SEPARATOR, '    ', 3, 0)
Token(KEYWORD, 'Keyword', 3, 4)
Token(SEPARATOR, '    ', 3, 11)
Token(ARGUMENT, 'argument', 3, 15)
Token(EOL, '\n', 3, 23)
Token(EOS, '', 3, 24)
Token(EOL, '\n', 4, 0)
Token(EOS, '', 4, 1)
```

The output shows the token type, value, line number and column offset. When finding tokens by their type, the constants in the *Token* class such as `Token.TESTCASE_NAME` and `Token.EOL` should be used instead the values of these constants like `'TESTCASE NAME'` and `'EOL'`. These values have changed slightly in Robot Framework 4.0 and they may change in the future as well.

The `EOL` tokens denote end of a line and they include the newline character and possible trailing spaces. The `EOS` tokens denote end of a logical statement. Typically a single line forms a statement, but when the `...` syntax is used for continuation, a statement spans multiple lines. In special cases a single line can also contain multiple statements.

Errors caused by unrecognized data such as non-existing section or setting names are handled during the tokenizing phase. Such errors are reported using tokens that have `ERROR` type and the actual error message in their `error` attribute. Syntax errors such as empty FOR loops are only handled when building the higher level model discussed below.

See the documentation of *`get_tokens()`* for details about different ways how to specify the data to be parsed, how to control should all tokens or only data tokens be returned, and should variables in keyword arguments and elsewhere be tokenized or not.

Parsing data to model

Data can be parsed to a higher level model by using *`get_model()`*, *`get_resource_model()`*, or *`get_init_model()`* functions depending on the type of the parsed file same way as when *parsing data to tokens*.

The model is represented as an abstract syntax tree (AST) implemented on top of Python's standard ast.AST class. To see how the model looks like, it is possible to use the ast.dump() function or the third-party astpretty module:

```
import ast
import astpretty
from robot.api.parsing import get_model

model = get_model('example.robot')
print(ast.dump(model, include_attributes=True))
print('-' * 72)
astpretty.pprint(model)
```

Running this code with the `example.robot` file from the previous section would produce so much output that it is not included here. If you are going to work with Robot Framework's AST, you are recommended to try that on your own.

Model objects

The model is build from nodes that are based ast.AST and further categorized to blocks and statements. Blocks can contain other blocks and statements as child nodes whereas statements only have tokens containing the actual data as *`Token`* instances. Both statements and blocks expose their position information via `lineno`, `col_offset`, `end_lineno` and `end_col_offset` attributes and some nodes have also other special attributes available.

Blocks:

- *`File`* (the root of the model)
- *`SettingSection`*
- *`VariableSection`*
- *`TestCaseSection`*
- *`KeywordSection`*
- *`CommentSection`*
- *`TestCase`*
- *`Keyword`*
- *`For`*
- *`If`*

Statements:

- *SectionHeader*
- *LibraryImport*
- *ResourceImport*
- *VariablesImport*
- *Documentation*
- *Metadata*
- *ForceTags*
- *DefaultTags*
- *SuiteSetup*
- *SuiteTeardown*
- *TestSetup*
- *TestTeardown*
- *TestTemplate*
- *TestTimeout*
- *Variable*
- *TestCaseName*
- *KeywordName*
- *Setup*
- *Teardown*
- *Tags*
- *Template*
- *Timeout*
- *Arguments*
- *Return*
- *KeywordCall*
- *TemplateArguments*
- *ForHeader*
- *IfHeader*
- *ElseIfHeader*
- *ElseHeader*
- *End*
- *Comment*
- *Error*
- *EmptyLine*

Inspecting model

The easiest way to inspect what data a model contains is implementing *ModelVisitor* and creating `visit_NodeName` to visit nodes with name `NodeName` as needed. The following example illustrates how to find what tests a certain test case file contains:

```
from robot.api.parsing import get_model, ModelVisitor


class TestNamePrinter(ModelVisitor):

    def visit_File(self, node):
        print(f"File '{node.source}' has following tests:")
        # Call `generic_visit` to visit also child nodes.
        self.generic_visit(node)

    def visit_TestCaseName(self, node):
        print(f"- {node.name} (on line {node.lineno})")


model = get_model('example.robot')
printer = TestNamePrinter()
printer.visit(model)
```

When the above code is run using the earlier `example.robot`, the output is this:

```
File 'example.robot' has following tests:
- Example (on line 2)
- Second example (on line 5)
```

Handling errors in model

All nodes in the model have `errors` attribute that contains possible errors the node has. These errors include syntax errors such as empty FOR loops or IF without a condition as well as errors caused by unrecognized data such as non-existing section or setting names.

Unrecognized data is handled already during the *tokenizing* phase. In the model such data is represented as *Error* nodes and their `errors` attribute contain error information got from the underlying `ERROR` tokens. Syntax errors do not create *Error* nodes, but instead the model has normal nodes such as *If* with errors in their `errors` attribute.

A simple way to go through the model and see are there errors is using the *ModelVisitor* discussed in the previous section:

```
class ErrorReporter(ModelVisitor):

    # Implement `generic_visit` to visit all nodes.
    def generic_visit(self, node):
        if node.errors:
            print(f'Error on line {node.lineno}:')
            for error in node.errors:
                print(f'- {error}')
        ModelVisitor.generic_visit(self, node)
```

Modifying data

Existing data the model contains can be modified simply by modifying values of the underlying tokens. If changes need to be saved, that is as easy as calling the *save()* method of the root model object. When just modifying token values, it is possible to still use *ModelVisitor* discussed in the above section. The next section discusses adding or removing nodes and then *ModelTransformer* should be used instead.

Modifications to tokens obviously require finding the tokens to be modified. The first step is finding nodes containing the tokens by implementing needed `visit_NodeName` methods. Then the exact token or tokens can be found using nodes' *get_token()* or *get_tokens()* methods. If only token values are needed, *get_value()* or *get_values()* can be used as a shortcut. First finding nodes and then the right tokens is illustrated by this keyword renaming example:

```
from robot.api.parsing import get_model, ModelVisitor, Token


class KeywordRenamer(ModelVisitor):

    def __init__(self, old_name, new_name):
        self.old_name = self.normalize(old_name)
        self.new_name = new_name

    def normalize(self, name):
        return name.lower().replace(' ', '').replace('_', '')

    def visit_KeywordName(self, node):
        '''Rename keyword definitions.'''
        if self.normalize(node.name) == self.old_name:
            token = node.get_token(Token.KEYWORD_NAME)
            token.value = self.new_name

    def visit_KeywordCall(self, node):
        '''Rename keyword usages.'''
        if self.normalize(node.keyword) == self.old_name:
            token = node.get_token(Token.KEYWORD)
            token.value = self.new_name


model = get_model('example.robot')
renamer = KeywordRenamer('Keyword', 'New Name')
renamer.visit(model)
model.save()
```

If you run the above example using the earlier `example.robot`, you can see that the `Keyword` keyword has been renamed to `New Name`. Notice that a real keyword renamer needed to take into account also keywords used with setups, teardowns and templates.

When token values are changed, column offset of the other tokens on same line are likely to be wrong. This does not affect saving the model or other typical usages, but if it is a problem then the caller needs to updated offsets separately.

Adding and removing nodes

Bigger changes to the model are somewhat more complicated than just modifying existing token values. When doing this kind of changes, *ModelTransformer* should be used instead of *ModelVisitor* that was discussed in the previous sections.

Removing nodes is relative easy and is accomplished by returning `None` from `visit_NodeName` methods. Remember to return the original node, or possibly a replacement node, from all of these methods when you do not want a node to be removed.

Adding nodes requires constructing needed *Model objects* and adding them to the model. The following example demonstrates both removing and adding nodes. If you run it against the earlier `example.robot`, you see that the first test gets a new keyword, the second test is removed, and settings section with documentation is added.

```
from robot.api.parsing import (
    get_model, Documentation, EmptyLine, KeywordCall,
    ModelTransformer, SettingSection, SectionHeader, Token
)


class TestModifier(ModelTransformer):

    def visit_TestCase(self, node):
        # The matched `TestCase` node is a block with `header` and
        # `body` attributes. `header` is a statement with familiar
        # `get_token` and `get_value` methods for getting certain
        # tokens or their value.
        name = node.header.get_value(Token.TESTCASE_NAME)
        # Returning `None` drops the node altogether i.e. removes
        # this test.
        if name == 'Second example':
            return None
        # Construct new keyword call statement from tokens. See `visit_File`
        # below for an example creating statements using `from_params`.
        new_keyword = KeywordCall([
            Token(Token.SEPARATOR, '    '),
            Token(Token.KEYWORD, 'New Keyword'),
            Token(Token.SEPARATOR, '    '),
            Token(Token.ARGUMENT, 'xxx'),
            Token(Token.EOL)
        ])
        # Add the keyword call to test as the second item.
        node.body.insert(1, new_keyword)
        # No need to call `generic_visit` because we are not
        # modifying child nodes. The node itself must to be
        # returned to avoid dropping it.
        return node

    def visit_File(self, node):
        # Create settings section with documentation. Needed header and body
        # statements are created using `from_params` method. This is typically
        # more convenient than creating statements based on tokens like above.
        settings = SettingSection(
            header=SectionHeader.from_params(Token.SETTING_HEADER),
            body=[
                Documentation.from_params('This is a really\npowerful API!'),
                EmptyLine.from_params()
            ]
        )
        # Add settings to the beginning of the file.
        node.sections.insert(0, settings)
        # Call `generic_visit` to visit also child nodes.
        return self.generic_visit(node)
```

(continues on next page)

(continued from previous page)

```
model = get_model('example.robot')
TestModifier().visit(model)
model.save('modified.robot')
```

Executing model

It is possible to convert a parsed and possibly modified model into an executable *TestSuite* structure by using its *from_model()* class method. In this case the *get_model()* function should be given the `curdir` argument to get possible `${CURDIR}` variable resolved correctly.

```
from robot.api import TestSuite
from robot.api.parsing import get_model

model = get_model('example.robot', curdir='/home/robot/example')
# modify model as needed
suite = TestSuite.from_model(model)
suite.run()
```

For more details about executing the created *TestSuite* object, see the documentation of its *run()* method. Notice also that if you do not need to modify the parsed model, it is easier to get the executable suite by using the *from_file_system()* class method.

robot.conf package

Implements settings for both test execution and output processing.

This package implements *RobotSettings* and *RebotSettings* classes used internally by the framework. There should be no need to use these classes externally.

This package can be considered relatively stable. Aforementioned classes are likely to be rewritten at some point to be more convenient to use. Instantiating them is not likely to change, though.

Submodules

robot.conf.gatherfailed module

class robot.conf.gatherfailed.**GatherFailedTests**

Bases: *robot.model.visitor.SuiteVisitor*

visit_test(*test*)

Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)

Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)

Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

class robot.conf.gatherfailed.**GatherFailedSuites**
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.conf.gatherfailed.**gather_failed_tests**(*output*)

robot.conf.gatherfailed.**gather_failed_suites**(*output*)

robot.conf.settings module

class robot.conf.settings.**RobotSettings**(*options=None*, ***extra_options*)
Bases: robot.conf.settings._BaseSettings

get_rebot_settings()

listeners

debug_file

suite_config

randomize_seed

randomize_suites

randomize_tests

dry_run

exit_on_failure

exit_on_error

skipped_tags

skip_on_failure

skip_teardown_on_exit

console_output_config

console_type

console_width

console_markers

max_error_lines

pre_run_modifiers

run_empty_suite

variables

variable_files

extension

console_colors

critical_tags

flatten_keywords

log

log_level

output

output_directory

pre_rebot_modifiers

remove_keywords

report

rpa

split_log

statistics_config

status_rc

xunit

class robot.conf.settings.**RebotSettings**(*options=None*, ***extra_options*)

Bases: robot.conf.settings._BaseSettings

suite_config

log_config

report_config

merge

console_output_config

console_colors

critical_tags

flatten_keywords

log

log_level

output

output_directory

pre_rebot_modifiers

process_empty_suite

remove_keywords

report

rpa

split_log

statistics_config

status_rc

xunit

expand_keywords

robot.htmldata package

Package for writing output files in HTML format.

This package is considered stable but it is not part of the public API.

Submodules

robot.htmldata.htmlfilewriter module

class robot.htmldata.htmlfilewriter.**HtmlFileWriter**(*output*, *model_writer*)
Bases: object

write(*template*)

class robot.htmldata.htmlfilewriter.**ModelWriter**
Bases: robot.htmldata.htmlfilewriter._Writer

handles(*line*)

write(*line*)

class robot.htmldata.htmlfilewriter.**LineWriter**(*output*)
Bases: robot.htmldata.htmlfilewriter._Writer

handles(*line*)

write(*line*)

class robot.htmldata.htmlfilewriter.**GeneratorWriter**(*html_writer*)
Bases: robot.htmldata.htmlfilewriter._Writer

write(*line*)

handles(*line*)

class robot.htmldata.htmlfilewriter.**JsFileWriter**(*html_writer*, *base_dir*)
Bases: robot.htmldata.htmlfilewriter._InliningWriter

write(*line*)

handles(*line*)

class robot.htmldata.htmlfilewriter.**CssFileWriter**(*html_writer*, *base_dir*)
Bases: robot.htmldata.htmlfilewriter._InliningWriter

write(*line*)

handles(*line*)

robot.htmldata.jartemplate module

robot.htmldata.jsonwriter module

class robot.htmldata.jsonwriter.**JsonWriter**(*output*, *separator=''*)
Bases: object

write_json(*prefix*, *data*, *postfix=';\n'*, *mapping=None*, *separator=True*)

write(*string*, *postfix=';\n'*, *separator=True*)

class robot.htmldata.jsonwriter.**JsonDumper**(*output*)
Bases: object

dump(*data*, *mapping=None*)

write(*data*)

class robot.htmldata.jsonwriter.**StringDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

dump(*data*, *mapping*)

handles(*data*, *mapping*)

class robot.htmldata.jsonwriter.**IntegerDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

dump(*data*, *mapping*)

handles(*data*, *mapping*)

class robot.htmldata.jsonwriter.**DictDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

dump(*data*, *mapping*)

handles(*data*, *mapping*)

class robot.htmldata.jsonwriter.**TupleListDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

dump(*data*, *mapping*)

handles(*data*, *mapping*)

class robot.htmldata.jsonwriter.**MappingDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

handles(*data*, *mapping*)

dump(*data*, *mapping*)

class robot.htmldata.jsonwriter.**NoneDumper**(*jsondumper*)
Bases: robot.htmldata.jsonwriter._Dumper

handles(*data*, *mapping*)

dump(*data*, *mapping*)

robot.htmldata.normaltemplate module

class robot.htmldata.normaltemplate.**HtmlTemplate**(*filename*)
Bases: object

robot.htmldata.template module

robot.libdocpkg package

Implements the *Libdoc* tool.

The command line entry point and programmatic interface for Libdoc are provided by the separate *robot.libdoc* module.

This package is considered stable but it is not part of the public API.

Submodules

robot.libdocpkg.builder module

robot.libdocpkg.builder.**JavaDocBuilder**()

robot.libdocpkg.builder.**LibraryDocumentation**(*library_or_resource*, *name=None*, *version=None*, *doc_format=None*)

robot.libdocpkg.builder.**DocumentationBuilder**(*library_or_resource*)
Create a documentation builder for the specified library or resource.

The argument can be a path to a library, a resource file or to a spec file generated by Libdoc earlier. If the argument does not point to an existing file, it is expected to be the name of the library to be imported. If a resource file is to be imported from PYTHONPATH, then *ResourceDocBuilder* must be used explicitly instead.

robot.libdocpkg.consoleviewer module

class robot.libdocpkg.consoleviewer.**ConsoleViewer**(*libdoc*)
Bases: object

classmethod handles(*command*)

classmethod validate_command(*command*, *args*)

view(*command*, **args*)

list(**patterns*)

show(**names*)

version()

class robot.libdocpkg.consoleviewer.**KeywordMatcher**(*libdoc*)
Bases: object

search(*patterns*)

robot.libdocpkg.datatypes module

class robot.libdocpkg.datatypes.**EnumType**
Bases: object

class robot.libdocpkg.datatypes.**DataTypeCatalog**
Bases: object

enums

typed_dicts

update(*types*)

to_dictionary()

class robot.libdocpkg.datatypes.**TypedDictDoc**(*name=''*, *doc=''*, *items=None*, *type='TypedDict'*)
Bases: *robot.utils.sortable.Sortable*

classmethod from_TypedDict(*typed_dict*)

to_dictionary()

class robot.libdocpkg.datatypes.**EnumDoc**(*name=''*, *doc=''*, *members=None*, *type='Enum'*)
Bases: *robot.utils.sortable.Sortable*

classmethod from_Enum(*enum_type*)

to_dictionary()

robot.libdocpkg.htmlutils module

class robot.libdocpkg.htmlutils.**DocFormatter**(*keywords*, *data_types*, *introduction*, *doc_format='ROBOT'*)
Bases: object

html(*doc*, *intro=False*)

class robot.libdocpkg.htmlutils.**DocToHtml**(*doc_format*)
Bases: object

class robot.libdocpkg.htmlutils.**HtmlToText**
Bases: object

html_tags = {'b': '*', 'code': '``', 'div.*?': '', 'em': '_', 'i': '_', 'strong'

html_chars = {'&': '&', ''': "'", '>': '>', '<': '<', '"': '"

get_shortdoc_from_html(*doc*)

html_to_plain_text(*doc*)

robot.libdocpkg.htmlwriter module

class robot.libdocpkg.htmlwriter.**LibdocHtmlWriter**
Bases: object

write(*libdoc*, *output*)

class robot.libdocpkg.htmlwriter.**LibdocModelWriter**(*output*, *libdoc*)
Bases: *robot.htmldata.htmlfilewriter.ModelWriter*

write(*line*)

handles(*line*)

robot.libdocpkg.java9builder module

robot.libdocpkg.javabuilder module

class robot.libdocpkg.javabuilder.**JavaDocBuilder**
Bases: object

build(*path*)

robot.libdocpkg.javabuilder.**ClassDoc**(*path*)
Process the given Java source file and return ClassDoc instance.

Processing is done using com.sun.tools.javadoc APIs. Returned object implements com.sun.javadoc.ClassDoc interface: http://docs.oracle.com/javase/7/docs/jdk/api/javadoc/doclet/

robot.libdocpkg.jsonbuilder module

class robot.libdocpkg.jsonbuilder.**JsonDocBuilder**
Bases: object

build(*path*)

build_from_dict(*spec*)

robot.libdocpkg.jsonwriter module

class robot.libdocpkg.jsonwriter.**LibdocJsonWriter**
Bases: object

write(*libdoc*, *outfile*)

robot.libdocpkg.model module

class robot.libdocpkg.model.**LibraryDoc**(*name=''*, *doc=''*, *version=''*, *type='LIBRARY'*, *scope='TEST'*, *doc_format='ROBOT'*, *source=None*, *lineno=-1*)
Bases: object

doc

doc_format

inits

keywords

all_tags

save(*output=None*, *format='HTML'*)

convert_docs_to_html()

to_dictionary()

to_json(*indent=None*)

class robot.libdocpkg.model.**KeywordDoc**(*name=''*, *args=()*, *doc=''*, *shortdoc=''*, *tags=()*, *source=None*, *lineno=-1*, *parent=None*)
Bases: *robot.utils.sortable.Sortable*

shortdoc

deprecated

generate_shortdoc()

to_dictionary()

robot.libdocpkg.output module

class robot.libdocpkg.output.**LibdocOutput**(*output_path*, *format*)
Bases: object

robot.libdocpkg.robotbuilder module

class robot.libdocpkg.robotbuilder.**LibraryDocBuilder**
Bases: object

build(*library*)

class robot.libdocpkg.robotbuilder.**ResourceDocBuilder**
Bases: object

build(*path*)

class robot.libdocpkg.robotbuilder.**KeywordDocBuilder**(*resource=False*)
Bases: object

build_keywords(*lib*)

build_keyword(*kw*)

robot.libdocpkg.specbuilder module

class robot.libdocpkg.specbuilder.**SpecDocBuilder**
Bases: object

build(*path*)

robot.libdocpkg.writer module

robot.libdocpkg.writer.**LibdocWriter**(*format=None*)

robot.libdocpkg.xmlwriter module

class robot.libdocpkg.xmlwriter.**LibdocXmlWriter**
Bases: object

write(*libdoc*, *outfile*)

robot.libraries package

Package hosting Robot Framework standard test libraries.

Libraries are mainly used externally in the test data, but they can be also used by custom test libraries if there is a need. Especially the *BuiltIn* library is often useful when there is a need to interact with the framework.

Because libraries are documented using Robot Framework's own documentation syntax, the generated API docs are not that well formed. It is thus better to find the generated library documentations, for example, via the http://robotframework.org web site.

Submodules

robot.libraries.BuiltIn module

robot.libraries.BuiltIn.**run_keyword_variant**(*resolve*)

class robot.libraries.BuiltIn.**BuiltIn**

Bases: robot.libraries.BuiltIn._Verify, robot.libraries.BuiltIn._Converter, robot.libraries.BuiltIn._Variables, robot.libraries.BuiltIn._RunKeyword, robot.libraries.BuiltIn._Control, robot.libraries.BuiltIn._Misc

An always available standard library with often needed keywords.

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

== Table of contents ==

%TOC%

= HTML error messages =

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

= Evaluating expressions =

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

== Evaluation namespace ==

Expressions are evaluated using Python's [http://docs.python.org/library/functions.html#eval|eval] function so that all Python built-ins like len() and int() are available. In addition to that, all unrecognized variables are considered to be modules that are automatically imported. It is possible to use all available Python modules, including the standard modules and the installed third party modules.

Evaluate also allows configuring the execution namespace with a custom namespace and with custom modules to be imported. The latter functionality is useful in special cases where the automatic module import does not work such as when using nested modules like rootmod.submod or list comprehensions. See the documentation of the *Evaluate* keyword for mode details.

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

== Using variables ==

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

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

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

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

= Boolean arguments =

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

True examples:

False examples:

Considering strings `OFF` and `0` false is new in Robot Framework 3.1.

= Pattern matching =

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

== Glob patterns ==

Some keywords, for example *Should Match*, support so called [http://en.wikipedia.org/wiki/Glob_(programming)|glob patterns] where:

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

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

== Regular expressions ==

Some keywords, for example *Should Match Regexp*, support [http://en.wikipedia.org/wiki/Regular_expression|regular expressions] that are more powerful but also more complicated that glob patterns. The regular expression support is implemented using Python's [http://docs.python.org/library/re.html|re module] and its documentation should be consulted for more information about the syntax.

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

= Multiline string comparison =

Should Be Equal and *Should Be Equal As Strings* report the failures using [http://en.wikipedia.org/wiki/Diff_utility#Unified_format|unified diff format] if both strings have more than two lines.

Results in the following error message:

= String representations =

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

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

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

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

== str ==

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

== repr ==

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

There are also some enhancements compared to the standard `repr()`: - Bigger lists, dictionaries and other containers are pretty-printed so

that there is one item per row.

- On Python 2 the `u` prefix is omitted with Unicode strings and the `b` prefix is added to byte strings.

== ascii ==

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

- On Python 3 non-ASCII characters are escaped like `\xe4` instead of showing them as-is like `ä`. This makes it easier to see differences between Unicode characters that look the same but are not equal. This is how `repr()` works in Python 2.

- On Python 2 just uses the standard `repr()` meaning that Unicode strings get the `u` prefix and no `b` prefix is added to byte strings.
- Containers are not pretty-printed.

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

ROBOT_LIBRARY_VERSION = '4.0.3'

call_method(*object*, *method_name*, **args*, ***kwargs*)
Calls the named method of the given object with the provided arguments.

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

Possible equal signs in arguments must be escaped with a backslash like `\=`.

catenate(**items*)
Catenates the given items together and returns the resulted string.

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

comment(**messages*)
Displays the given messages in the log file as keyword arguments.

This keyword does nothing with the arguments it receives, but as they are visible in the log, this keyword can be used to display simple messages. Given arguments are ignored so thoroughly that they can even contain non-existing variables. If you are interested about variable values, you can use the *Log* or *Log Many* keywords.

continue_for_loop()
Skips the current for loop iteration and continues from the next.

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

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

continue_for_loop_if(*condition*)
Skips the current for loop iteration if the `condition` is true.

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

convert_to_binary(*item*, *base=None*, *prefix=None*, *length=None*)
Converts the given item to a binary string.

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

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

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

convert_to_boolean(*item*)
Converts the given item to Boolean true or false.

Handles strings `True` and `False` (case-insensitive) as expected, otherwise returns item's [http://docs.python.org/library/stdtypes.html#truth|truth value] using Python's `bool()` method.

convert_to_bytes(*input*, *input_type='text'*)
Converts the given `input` to bytes according to the `input_type`.

Valid input types are listed below:

- `text`: Converts text to bytes character by character. All characters with ordinal below 256 can be used and are converted to bytes with same values. Many characters are easiest to represent using escapes like `\x00` or `\xff`. Supports both Unicode strings and bytes.
- `int`: Converts integers separated by spaces to bytes. Similarly as with *Convert To Integer*, it is possible to use binary, octal, or hex values by prefixing the values with `0b`, `0o`, or `0x`, respectively.
- `hex`: Converts hexadecimal values to bytes. Single byte is always two characters long (e.g. `01` or `FF`). Spaces are ignored and can be used freely as a visual separator.
- `bin`: Converts binary values to bytes. Single byte is always eight characters long (e.g. `00001010`). Spaces are ignored and can be used freely as a visual separator.

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

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

convert_to_hex(*item*, *base=None*, *prefix=None*, *length=None*, *lowercase=False*)
Converts the given item to a hexadecimal string.

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

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

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

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

convert_to_integer(*item*, *base=None*)
Converts the given item to an integer number.

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

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

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

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

convert_to_number(*item*, *precision=None*)
Converts the given item to a floating point number.

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

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

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

If you want to avoid possible problems with floating point numbers, you can implement custom keywords using Python's [http://docs.python.org/library/decimal.html|decimal] or [http://docs.python.org/library/fractions.html|fractions] modules.

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

convert_to_octal(*item*, *base=None*, *prefix=None*, *length=None*)
Converts the given item to an octal string.

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

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

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

convert_to_string(*item*)
Converts the given item to a Unicode string.

Strings are also [http://www.macchiato.com/unicode/nfc-faq| NFC normalized].

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

create_dictionary(**items*)
Creates and returns a dictionary based on the given `items`.

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

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

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

create_list(**items*)
Returns a list containing given items.

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

evaluate(*expression*, *modules=None*, *namespace=None*)
Evaluates the given expression in Python and returns the result.

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

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

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

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

Starting from Robot Framework 3.2, modules used in the expression are imported automatically. There are, however, two cases where they need to be explicitly specified using the `modules` argument:

- When nested modules like `rootmod.submod` are implemented so that the root module does not automatically import sub modules. This is illustrated by the `selenium.webdriver` example below.
- When using a module in the expression part of a list comprehension. This is illustrated by the `json` example below.

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

exit_for_loop()
Stops executing the enclosing for loop.

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

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

exit_for_loop_if(*condition*)
Stops executing the enclosing for loop if the `condition` is true.

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

fail(*msg=None*, **tags*)
Fails the test with the given message and optionally alters its tags.

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

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

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

fatal_error(*msg=None*)
Stops the whole test execution.

The test or suite where this keyword is used fails with the provided message, and subsequent tests fail with a canned message. Possible teardowns will nevertheless be executed.

See *Fail* if you only want to stop one test case unconditionally.

get_count(*container*, *item*)
Returns and logs how many times `item` is found from `container`.

This keyword works with Python strings and lists and all objects that either have `count` method or can be converted to Python lists.

get_length(*item*)
Returns and logs the length of the given item as an integer.

The item can be anything that has a length, for example, a string, a list, or a mapping. The keyword first tries to get the length with the Python function `len`, which calls the item's `__len__` method internally. If that fails, the keyword tries to call the item's possible `length` and `size` methods directly. The final attempt is trying to get the value of the item's `length` attribute. If all these attempts are unsuccessful, the keyword fails.

See also *Length Should Be*, *Should Be Empty* and *Should Not Be Empty*.

get_library_instance (*name=None*, *all=False*)
Returns the currently active instance of the specified test library.

This keyword makes it easy for test libraries to interact with other test libraries that have state. This is illustrated by the Python example below:

It is also possible to use this keyword in the test data and pass the returned library instance to another keyword. If a library is imported with a custom name, the `name` used to get the instance must be that name and not the original library name.

If the optional argument `all` is given a true value, then a dictionary mapping all library names to instances will be returned.

get_time (*format='timestamp'*, *time_='NOW'*)
Returns the given time in the requested format.

NOTE: DateTime library contains much more flexible keywords for getting the current date and time and for date and time handling in general.

How time is returned is determined based on the given `format` string as follows. Note that all checks are case-insensitive.

1) If `format` contains the word `epoch`, the time is returned in seconds after the UNIX epoch (1970-01-01 00:00:00 UTC). The return value is always an integer.
2) If `format` contains any of the words `year`, `month`, `day`, `hour`, `min`, or `sec`, only the selected parts are returned. The order of the returned parts is always the one in the previous sentence and the order of words in `format` is not significant. The parts are returned as zero-padded strings (e.g. May -> `05`).
3) Otherwise (and by default) the time is returned as a timestamp string in the format `2006-02-24 15:08:31`.

By default this keyword returns the current local time, but that can be altered using `time` argument as explained below. Note that all checks involving strings are case-insensitive.

1) If `time` is a number, or a string that can be converted to a number, it is interpreted as seconds since the UNIX epoch. This documentation was originally written about 1177654467 seconds after the epoch.
2) If `time` is a timestamp, that time will be used. Valid timestamp formats are `YYYY-MM-DD hh:mm:ss` and `YYYYMMDD hhmmss`.
3) If `time` is equal to `NOW` (default), the current local time is used.
4) If `time` is equal to `UTC`, the current time in [http://en.wikipedia.org/wiki/Coordinated_Universal_Time|UTC] is used.
5) If `time` is in the format like `NOW - 1 day` or `UTC + 1 hour 30 min`, the current local/UTC time plus/minus the time specified with the time string is used. The time string format is described in an appendix of Robot Framework User Guide.

UTC time is 2006-03-29 12:06:21):

get_variable_value(*name*, *default=None*)
Returns variable value or `default` if the variable does not exist.

The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*.

See *Set Variable If* for another keyword to set variables dynamically.

get_variables(*no_decoration=False*)
Returns a dictionary containing all variables in the current scope.

Variables are returned as a special dictionary that allows accessing variables in space, case, and underscore insensitive manner similarly as accessing variables in the test data. This dictionary supports all same operations as normal Python dictionaries and, for example, Collections library can be used to access or modify it. Modifying the returned dictionary has no effect on the variables available in the current scope.

By default variables are returned with `${}`, `@{}` or `&{}` decoration based on variable types. Giving a true value (see *Boolean arguments*) to the optional argument `no_decoration` will return the variables without the decoration.

import_library(*name*, **args*)
Imports a library with the given name and optional arguments.

This functionality allows dynamic importing of libraries while tests are running. That may be necessary, if the library itself is dynamic and not yet available when test data is processed. In a normal case, libraries should be imported using the Library setting in the Setting table.

This keyword supports importing libraries both using library names and physical paths. When paths are used, they must be given in absolute format or found from [http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pythonpath-jythonpath-and-ironpythonpath| search path]. Forward slashes can be used as path separators in all operating systems.

It is possible to pass arguments to the imported library and also named argument syntax works if the library supports it. `WITH NAME` syntax can be used to give a custom name to the imported library.

import_resource(*path*)
Imports a resource file with the given path.

Resources imported with this keyword are set into the test suite scope similarly when importing them in the Setting table using the Resource setting.

The given path must be absolute or found from [http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pythonpath-jythonpath-and-ironpythonpath| search path]. Forward slashes can be used as path separator regardless the operating system.

import_variables(*path*, **args*)
Imports a variable file with the given path and optional arguments.

Variables imported with this keyword are set into the test suite scope similarly when importing them in the Setting table using the Variables setting. These variables override possible existing variables with the same names. This functionality can thus be used to import new variables, for example, for each test in a test suite.

The given path must be absolute or found from [http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#pythonpath-jythonpath-and-ironpythonpath| search path]. Forward slashes can be used as path separator regardless the operating system.

keyword_should_exist(*name*, *msg=None*)
Fails unless the given keyword exists in the current scope.

Fails also if there are more than one keywords with the same name. Works both with the short name (e.g. `Log`) and the full name (e.g. `BuiltIn.Log`).

The default error message can be overridden with the `msg` argument.

See also *Variable Should Exist*.

length_should_be(*item*, *length*, *msg=None*)
Verifies that the length of the given item is correct.

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

log(*message*, *level='INFO'*, *html=False*, *console=False*, *repr=False*, *formatter='str'*)
Logs the given message with the given level.

Valid levels are TRACE, DEBUG, INFO (default), HTML, WARN, and ERROR. Messages below the current active log level are ignored. See *Set Log Level* keyword and `--loglevel` command line option for more details about setting the level.

Messages logged with the WARN or ERROR levels will be automatically visible also in the console and in the Test Execution Errors section in the log file.

If the `html` argument is given a true value (see *Boolean arguments*), the message will be considered HTML and special characters such as < are not escaped. For example, logging `<img src="image.png">` creates an image when `html` is true, but otherwise the message is that exact string. An alternative to using the `html` argument is using the HTML pseudo log level. It logs the message as HTML using the INFO level.

If the `console` argument is true, the message will be written to the console where test execution was started from in addition to the log file. This keyword always uses the standard output stream and adds a newline after the written message. Use *Log To Console* instead if either of these is undesirable,

The `formatter` argument controls how to format the string representation of the message. Possible values are `str` (default), `repr` and `ascii`, and they work similarly to Python built-in functions with same names. When using `repr`, bigger lists, dictionaries and other containers are also pretty-printed so that there is one item per row. For more details see *String representations*. This is a new feature in Robot Framework 3.1.2.

The old way to control string representation was using the `repr` argument, and `repr=True` is still equivalent to using `formatter=repr`. The `repr` argument will be deprecated in the future, though, and using `formatter` is thus recommended.

See *Log Many* if you want to log multiple messages in one go, and *Log To Console* if you only want to write to the console.

log_many(**messages*)
Logs the given messages as separate entries using the INFO level.

Supports also logging list and dictionary variable items individually.

See *Log* and *Log To Console* keywords if you want to use alternative log levels, use HTML, or log to the console.

log_to_console(*message*, *stream='STDOUT'*, *no_newline=False*)
Logs the given message to the console.

By default uses the standard output stream. Using the standard error stream is possibly by giving the `stream` argument value `STDERR` (case-insensitive).

By default appends a newline to the logged message. This can be disabled by giving the `no_newline` argument a true value (see *Boolean arguments*).

This keyword does not log the message to the normal log file. Use *Log* keyword, possibly with argument `console`, if that is desired.

log_variables(*level='INFO'*)
Logs all variables in the current scope with given log level.

no_operation()
Does absolutely nothing.

pass_execution(*message*, **tags*)
Skips rest of the current test, setup, or teardown with PASS status.

This keyword can be used anywhere in the test data, but the place where used affects the behavior:

- When used in any setup or teardown (suite, test or keyword), passes that setup or teardown. Possible keyword teardowns of the started keywords are executed. Does not affect execution or statuses otherwise.
- When used in a test outside setup or teardown, passes that particular test case. Possible test and keyword teardowns are executed.

Possible continuable failures before this keyword is used, as well as failures in executed teardowns, will fail the execution.

It is mandatory to give a message explaining why execution was passed. By default the message is considered plain text, but starting it with `*HTML*` allows using HTML formatting.

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

This keyword is typically wrapped to some other keyword, such as *Run Keyword If*, to pass based on a condition. The most common case can be handled also with *Pass Execution If*:

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

pass_execution_if(*condition*, *message*, **tags*)
Conditionally skips rest of the current test, setup, or teardown with PASS status.

A wrapper for *Pass Execution* to skip rest of the current test, setup or teardown based the given `condition`. The condition is evaluated similarly as with *Should Be True* keyword, and `message` and `*tags` have same semantics as with *Pass Execution*.

regexp_escape(**patterns*)
Returns each argument string escaped for use as a regular expression.

This keyword can be used to escape strings to be used with *Should Match Regexp* and *Should Not Match Regexp* keywords.

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

reload_library(*name_or_instance*)
Rechecks what keywords the specified library provides.

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

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

remove_tags(**tags*)
Removes given `tags` from the current test or all tests in a suite.

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

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

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

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

repeat_keyword(*repeat*, *name*, **args*)
Executes the specified keyword multiple times.

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

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

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

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

replace_variables(*text*)
Replaces variables in the given text with their current values.

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

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

return_from_keyword(**return_values*)
Returns from the enclosing user keyword.

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

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

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

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

return_from_keyword_if(*condition*, **return_values*)
Returns from the enclosing user keyword if `condition` is true.

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

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

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

run_keyword(*name*, **args*)
Executes the given keyword with the given arguments.

Because the name of the keyword to execute is given as an argument, it can be a variable and thus set dynamically, e.g. from a return value of another keyword or from the command line.

run_keyword_and_continue_on_failure(*name*, **args*)
Runs the keyword and continues execution even if a failure occurs.

The keyword name and arguments work as with *Run Keyword*.

The execution is not continued if the failure is caused by invalid syntax, timeout, or fatal exception.

run_keyword_and_expect_error(*expected_error*, *name*, **args*)
Runs the keyword and checks that the expected error occurred.

The keyword to execute and its arguments are specified using `name` and `*args` exactly like with *Run Keyword*.

The expected error must be given in the same format as in Robot Framework reports. By default it is interpreted as a glob pattern with `*`, `?` and `[chars]` as wildcards, but starting from Robot Framework 3.1 that can be changed by using various prefixes explained in the table below. Prefixes are case-sensitive and they must be separated from the actual message with a colon and an optional space like `PREFIX: Message` or `PREFIX:Message`.

See the *Pattern matching* section for more information about glob patterns and regular expressions.

If the expected error occurs, the error message is returned and it can be further processed or tested if needed. If there is no error, or the error does not match the expected error, this keyword fails.

Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword.

run_keyword_and_ignore_error(*name*, **args*)
Runs the given keyword with the given arguments and ignores possible error.

This keyword returns two values, so that the first is either string `PASS` or `FAIL`, depending on the status of the executed keyword. The second value is either the return value of the keyword or the received error message. See *Run Keyword And Return Status* If you are only interested in the execution status.

The keyword name and arguments work as in *Run Keyword*. See *Run Keyword If* for a usage example.

Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword. Otherwise this keyword itself never fails.

run_keyword_and_return(*name*, **args*)
Runs the specified keyword and returns from the enclosing user keyword.

The keyword to execute is defined with `name` and `*args` exactly like with *Run Keyword*. After running the keyword, returns from the enclosing user keyword and passes possible return value from the executed keyword further. Returning from a keyword has exactly same semantics as with *Return From Keyword*.

Use *Run Keyword And Return If* if you want to run keyword and return based on a condition.

run_keyword_and_return_if(*condition*, *name*, **args*)
Runs the specified keyword and returns from the enclosing user keyword.

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

Use *Return From Keyword If* if you want to return a certain value based on a condition.

run_keyword_and_return_status(*name*, **args*)
Runs the given keyword with given arguments and returns the status as a Boolean value.

This keyword returns Boolean `True` if the keyword that is executed succeeds and `False` if it fails. This is useful, for example, in combination with *Run Keyword If*. If you are interested in the error message or return value, use *Run Keyword And Ignore Error* instead.

The keyword name and arguments work as in *Run Keyword*.

Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword. Otherwise this keyword itself never fails.

run_keyword_and_warn_on_failure(*name*, **args*)
Runs the specified keyword logs a warning if the keyword fails.

This keyword is similar to *Run Keyword And Ignore Error* but if the executed keyword fails, the error message is logged as a warning to make it more visible. Returns status and possible return value or error message exactly like *Run Keyword And Ignore Error* does.

Errors caused by invalid syntax, timeouts, or fatal exceptions are not caught by this keyword. Otherwise this keyword itself never fails.

New in Robot Framework 4.0.

run_keyword_if(*condition*, *name*, **args*)
Runs the given keyword with the given arguments, if `condition` is true.

NOTE: Robot Framework 4.0 introduced built-in IF/ELSE support and using that is generally recommended over using this keyword.

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

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

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

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

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

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

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

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

Python's [http://docs.python.org/library/os.html|os] and [http://docs.python.org/library/sys.html|sys] modules are automatically imported when evaluating the `condition`. Attributes they contain can thus be used in the condition:

run_keyword_if_all_critical_tests_passed(*name*, **args*)
DEPRECATED. Use *BuiltIn.Run Keyword If All Tests Passed* instead.

run_keyword_if_all_tests_passed(*name*, **args*)
Runs the given keyword with the given arguments, if all tests passed.

This keyword can only be used in a suite teardown. Trying to use it anywhere else results in an error.

Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.

run_keyword_if_any_critical_tests_failed(*name*, **args*)
DEPRECATED. Use *BuiltIn.Run Keyword If Any Tests Failed* instead.

run_keyword_if_any_tests_failed(*name*, **args*)
Runs the given keyword with the given arguments, if one or more tests failed.

This keyword can only be used in a suite teardown. Trying to use it anywhere else results in an error.

Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.

run_keyword_if_test_failed(*name*, **args*)
Runs the given keyword with the given arguments, if the test failed.

This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error.

Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.

run_keyword_if_test_passed(*name*, **args*)
Runs the given keyword with the given arguments, if the test passed.

This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error.

Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.

run_keyword_if_timeout_occurred(*name*, **args*)
Runs the given keyword if either a test or a keyword timeout has occurred.

This keyword can only be used in a test teardown. Trying to use it anywhere else results in an error.

Otherwise, this keyword works exactly like *Run Keyword*, see its documentation for more details.

run_keyword_unless(*condition*, *name*, **args*)
Runs the given keyword with the given arguments if `condition` is false.

See *Run Keyword If* for more information and an example. Notice that this keyword does not support `ELSE` or `ELSE IF` branches like *Run Keyword If* does, though.

run_keywords(**keywords*)
Executes all the given keywords in a sequence.

This keyword is mainly useful in setups and teardowns when they need to take care of multiple actions and creating a new higher level user keyword would be an overkill.

By default all arguments are expected to be keywords to be executed.

Keywords can also be run with arguments using upper case `AND` as a separator between keywords. The keywords are executed so that the first argument is the first keyword and proceeding arguments until the first `AND` are arguments to it. First argument after the first `AND` is the second keyword and proceeding arguments until the next `AND` are its arguments. And so on.

Notice that the `AND` control argument must be used explicitly and cannot itself come from a variable. If you need to use literal `AND` string as argument, you can either use variables or escape it with a backslash like `\AND`.

set_global_variable(*name*, **values*)
Makes a variable available globally in all tests and suites.

Variables set with this keyword are globally available in all subsequent test suites, test cases and user keywords. Also variables in variable tables are overridden. Variables assigned locally based on keyword return values or by using *Set Test Variable* and *Set Suite Variable* override these variables in that scope, but the global value is not changed in those cases.

In practice setting variables with this keyword has the same effect as using command line options `--variable` and `--variablefile`. Because this keyword can change variables everywhere, it should be used with care.

See *Set Suite Variable* for more information and examples.

set_library_search_order(**search_order*)
Sets the resolution order to use when a name matches multiple keywords.

The library search order is used to resolve conflicts when a keyword name in the test data matches multiple keywords. The first library (or resource, see below) containing the keyword is selected and that keyword implementation used. If the keyword is not found from any library (or resource), test executing fails the same way as when the search order is not set.

When this keyword is used, there is no need to use the long `LibraryName.Keyword Name` notation. For example, instead of having

you can have

This keyword can be used also to set the order of keywords in different resource files. In this case resource names must be given without paths or extensions like:

NOTE: - The search order is valid only in the suite where this keywords is used. - Keywords in resources always have higher priority than

keywords in libraries regardless the search order.

- The old order is returned and can be used to reset the search order later.
- Library and resource names in the search order are both case and space insensitive.

set_local_variable(*name*, **values*)
Makes a variable available everywhere within the local scope.

Variables set with this keyword are available within the local scope of the currently executed test case or in the local scope of the keyword in which they are defined. For example, if you set a variable in a user keyword, it is available only in that keyword. Other test cases or keywords will not see variables set with this keyword.

This keyword is equivalent to a normal variable assignment based on a keyword return value.

is equivalent with

This keyword will provide the option of setting local variables inside keywords like *Run Keyword If*, *Run Keyword And Return If*, *Run Keyword Unless* which until now was not possible by using *Set Variable*.

It will also be possible to use this keyword from external libraries that want to set local variables.

New in Robot Framework 3.2.

set_log_level (*level*)
Sets the log threshold to the specified level and returns the old level.

Messages below the level will not logged. The default logging level is INFO, but it can be overridden with the command line option `--loglevel`.

The available levels: TRACE, DEBUG, INFO (default), WARN, ERROR and NONE (no logging).

set_suite_documentation (*doc*, *append=False*, *top=False*)
Sets documentation for the current test suite.

By default the possible existing documentation is overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword.

This keyword sets the documentation of the current suite by default. If the optional `top` argument is given a true value (see *Boolean arguments*), the documentation of the top level suite is altered instead.

The documentation of the current suite is available as a built-in variable `${SUITE DOCUMENTATION}`.

set_suite_metadata (*name*, *value*, *append=False*, *top=False*)
Sets metadata for the current test suite.

By default possible existing metadata values are overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword.

This keyword sets the metadata of the current suite by default. If the optional `top` argument is given a true value (see *Boolean arguments*), the metadata of the top level suite is altered instead.

The metadata of the current suite is available as a built-in variable `${SUITE METADATA}` in a Python dictionary. Notice that modifying this variable directly has no effect on the actual metadata the suite has.

set_suite_variable (*name*, **values*)
Makes a variable available everywhere within the scope of the current suite.

Variables set with this keyword are available everywhere within the scope of the currently executed test suite. Setting variables with this keyword thus has the same effect as creating them using the Variable table in the test data file or importing them from variable files.

Possible child test suites do not see variables set with this keyword by default, but that can be controlled by using `children=<option>` as the last argument. If the specified `<option>` given a true value (see *Boolean arguments*), the variable is set also to the child suites. Parent and sibling suites will never see variables set with this keyword.

The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format as `\${NAME}` or `$NAME`. Variable value can be given using the same syntax as when variables are created in the Variable table.

If a variable already exists within the new scope, its value will be overwritten. Otherwise a new variable is created. If a variable already exists within the current scope, the value can be left empty and the variable within the new scope gets the value within the current scope.

To override an existing value with an empty value, use built-in variables `${EMPTY}`, `@{EMPTY}` or `&{EMPTY}`:

NOTE: If the variable has value which itself is a variable (escaped or not), you must always use the escaped format to set the variable:

This limitation applies also to *Set Test Variable*, *Set Global Variable*, *Variable Should Exist*, *Variable Should Not Exist* and *Get Variable Value* keywords.

set_tags (**tags*)
Adds given `tags` for the current test or all tests in a suite.

When this keyword is used inside a test case, that test gets the specified tags and other tests are not affected.

If this keyword is used in a suite setup, all test cases in that suite, recursively, gets the given tags. It is a failure to use this keyword in a suite teardown.

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

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

set_task_variable(*name*, **values*)
Makes a variable available everywhere within the scope of the current task.

This is an alias for *Set Test Variable* that is more applicable when creating tasks, not tests. New in Robot Framework 3.1.

set_test_documentation(*doc*, *append=False*)
Sets documentation for the current test case.

By default the possible existing documentation is overwritten, but this can be changed using the optional `append` argument similarly as with *Set Test Message* keyword.

The current test documentation is available as a built-in variable `${TEST DOCUMENTATION}`. This keyword can not be used in suite setup or suite teardown.

set_test_message(*message*, *append=False*)
Sets message for the current test case.

If the optional `append` argument is given a true value (see *Boolean arguments*), the given `message` is added after the possible earlier message by joining the messages with a space.

In test teardown this keyword can alter the possible failure message, but otherwise failures override messages set by this keyword. Notice that in teardown the message is available as a built-in variable `${TEST MESSAGE}`.

It is possible to use HTML format in the message by starting the message with `*HTML*`.

This keyword can not be used in suite setup or suite teardown.

set_test_variable(*name*, **values*)
Makes a variable available everywhere within the scope of the current test.

Variables set with this keyword are available everywhere within the scope of the currently executed test case. For example, if you set a variable in a user keyword, it is available both in the test case level and also in all other user keywords used in the current test. Other test cases will not see variables set with this keyword.

See *Set Suite Variable* for more information and examples.

set_variable(**values*)
Returns the given values which can then be assigned to a variables.

This keyword is mainly used for setting scalar variables. Additionally it can be used for converting a scalar variable containing a list to a list variable or to multiple scalar variables. It is recommended to use *Create List* when creating new lists.

Variables created with this keyword are available only in the scope where they are created. See *Set Global Variable*, *Set Test Variable* and *Set Suite Variable* for information on how to set variables so that they are available also in a larger scope.

set_variable_if(*condition*, **values*)
Sets variable based on the given condition.

The basic usage is giving a condition and two values. The given condition is first evaluated the same way as with the *Should Be True* keyword. If the condition is true, then the first value is returned, and otherwise

the second value is returned. The second value can also be omitted, in which case it has a default value None. This usage is illustrated in the examples below, where `${rc}` is assumed to be zero.

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

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

should_be_empty (*item*, *msg=None*)
Verifies that the given item is empty.

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

should_be_equal (*first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *formatter='str'*, *strip_spaces=False*)
Fails if the given objects are unequal.

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

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

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

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

`formatter` is new in Robot Framework 3.1.2 and `strip_spaces` is new in Robot Framework 4.0.

should_be_equal_as_integers (*first*, *second*, *msg=None*, *values=True*, *base=None*)
Fails if objects are unequal after converting them to integers.

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

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

should_be_equal_as_numbers (*first*, *second*, *msg=None*, *values=True*, *precision=6*)
Fails if objects are unequal after converting them to real numbers.

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

As discussed in the documentation of *Convert To Number*, machines generally cannot store floating point numbers accurately. Because of this limitation, comparing floats for equality is problematic and a correct approach to use depends on the context. This keyword uses a very naive approach of rounding the numbers before comparing them, which is both prone to rounding errors and does not work very well if

numbers are really big or small. For more information about comparing floats, and ideas on how to implement your own context specific comparison algorithm, see http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/.

If you want to avoid possible problems with floating point numbers, you can implement custom keywords using Python's [http://docs.python.org/library/decimal.html|decimal] or [http://docs.python.org/library/fractions.html|fractions] modules.

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

should_be_equal_as_strings (*first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*, *formatter='str'*)

Fails if objects are unequal after converting them to strings.

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

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

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

Strings are always [http://www.macchiato.com/unicode/nfc-faq| NFC normalized].

`formatter` is new in Robot Framework 3.1.2 and `strip_spaces` is new in Robot Framework 4.0.

should_be_true (*condition*, *msg=None*)

Fails if the given condition is not true.

If `condition` is a string (e.g. `${rc} < 10`), it is evaluated as a Python expression as explained in *Evaluating expressions* and the keyword status is decided based on the result. If a non-string item is given, the status is got directly from its [http://docs.python.org/library/stdtypes.html#truth|truth value].

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

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

Should Be True automatically imports Python's [http://docs.python.org/library/os.html|os] and [http://docs.python.org/library/sys.html|sys] modules that contain several useful attributes:

should_contain (*container*, *item*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)

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

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

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

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

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

`strip_spaces` is new in Robot Framework 4.0.

should_contain_any(*container*, **items*, ***configuration*)
Fails if `container` does not contain any of the `*items`.

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

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

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

should_contain_x_times(*container*, *item*, *count*, *msg=None*, *ignore_case=False*, *strip_spaces=False*)
Fails if `container` does not contain `item` `count` times.

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

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

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

`strip_spaces` is new in Robot Framework 4.0.

should_end_with(*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if the string `str1` does not end with the string `str2`.

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

should_match(*string*, *pattern*, *msg=None*, *values=True*, *ignore_case=False*)
Fails if the given `string` does not match the given `pattern`.

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

If `ignore_case` is given a true value (see *Boolean arguments*) and compared items are strings, it indicates that comparison should be case-insensitive.

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

should_match_regexp(*string*, *pattern*, *msg=None*, *values=True*)
Fails if `string` does not match `pattern` as a regular expression.

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

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

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

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

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

should_not_be_empty(*item*, *msg=None*)
Verifies that the given item is not empty.

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

should_not_be_equal(*first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if the given objects are equal.

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

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

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

`strip_spaces` is new in Robot Framework 4.0.

should_not_be_equal_as_integers(*first*, *second*, *msg=None*, *values=True*, *base=None*)
Fails if objects are equal after converting them to integers.

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

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

See *Should Be Equal As Integers* for some usage examples.

should_not_be_equal_as_numbers(*first*, *second*, *msg=None*, *values=True*, *precision=6*)
Fails if objects are equal after converting them to real numbers.

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

See *Should Be Equal As Numbers* for examples on how to use `precision` and why it does not always work as expected. See also *Should Be Equal* for an explanation on how to override the default error message with `msg` and `values`.

should_not_be_equal_as_strings(*first*, *second*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if objects are equal after converting them to strings.

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

If `ignore_case` is given a true value (see *Boolean arguments*), comparison is done case-insensitively.

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

Strings are always [http://www.macchiato.com/unicode/nfc-faql NFC normalized].

`strip_spaces` is new in Robot Framework 4.0.

should_not_be_true(*condition*, *msg=None*)
Fails if the given condition is true.

See *Should Be True* for details about how `condition` is evaluated and how `msg` can be used to override the default error message.

should_not_contain(*container*, *item*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if `container` contains `item` one or more times.

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

See *Should Be Equal* for an explanation on how to override the default error message with arguments `msg` and `values`. `ignore_case` has exactly the same semantics as with *Should Contain*.

If `strip_spaces` is given a true value (see *Boolean arguments*) and both arguments are strings, the comparison is done without leading and trailing spaces. If `strip_spaces` is given a string value `LEADING` or `TRAILING` (case-insensitive), the comparison is done without leading or trailing spaces, respectively.

`strip_spaces` is new in Robot Framework 4.0.

should_not_contain_any(*container*, **items*, ***configuration*)
Fails if `container` contains one or more of the `*items`.

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

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

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

should_not_end_with(*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if the string `str1` ends with the string `str2`.

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

should_not_match(*string*, *pattern*, *msg=None*, *values=True*, *ignore_case=False*)
Fails if the given `string` matches the given `pattern`.

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

If `ignore_case` is given a true value (see *Boolean arguments*), the comparison is case-insensitive.

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

should_not_match_regexp(*string*, *pattern*, *msg=None*, *values=True*)
Fails if `string` matches `pattern` as a regular expression.

See *Should Match Regexp* for more information about arguments.

should_not_start_with(*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if the string `str1` starts with the string `str2`.

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

should_start_with(*str1*, *str2*, *msg=None*, *values=True*, *ignore_case=False*, *strip_spaces=False*)
Fails if the string `str1` does not start with the string `str2`.

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

skip(*msg='Skipped with Skip keyword.'*)
Skips the rest of the current test.

Skips the remaining keywords in the current test and sets the given message to the test. If the test has teardown, it will be executed.

skip_if(*condition*, *msg=None*)
Skips the rest of the current test if the `condition` is True.

Skips the remaining keywords in the current test and sets the given message to the test. If `msg` is not given, the `condition` will be used as the message. If the test has teardown, it will be executed.

If the `condition` evaluates to False, does nothing.

sleep(*time_*, *reason=None*)
Pauses the test executed for the given time.

`time` may be either a number or a time string. Time strings are in a format such as `1 day 2 hours 3 minutes 4 seconds 5milliseconds` or `1d 2h 3m 4s 5ms`, and they are fully explained in an appendix of Robot Framework User Guide. Optional *reason* can be used to explain why sleeping is necessary. Both the time slept and the reason are logged.

variable_should_exist(*name*, *msg=None*)
Fails unless the given variable exists within the current scope.

The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*.

The default error message can be overridden with the `msg` argument.

See also *Variable Should Not Exist* and *Keyword Should Exist*.

variable_should_not_exist(*name*, *msg=None*)
Fails if the given variable exists within the current scope.

The name of the variable can be given either as a normal variable name (e.g. `${NAME}`) or in escaped format (e.g. `\${NAME}`). Notice that the former has some limitations explained in *Set Suite Variable*.

The default error message can be overridden with the `msg` argument.

See also *Variable Should Exist* and *Keyword Should Exist*.

wait_until_keyword_succeeds(*retry*, *retry_interval*, *name*, **args*)
Runs the specified keyword and retries if it fails.

`name` and `args` define the keyword that is executed similarly as with *Run Keyword*. How long to retry running the keyword is defined using `retry` argument either as timeout or count. `retry_interval` is the time to wait before trying to run the keyword again after the previous run has failed.

If `retry` is given as timeout, it must be in Robot Framework's time format (e.g. `1 minute`, `2 min 3 s`, `4.5`) that is explained in an appendix of Robot Framework User Guide. If it is given as count, it must have `times` or `x` postfix (e.g. `5 times`, `10 x`). `retry_interval` must always be given in Robot Framework's time format.

If the keyword does not succeed regardless of retries, this keyword fails. If the executed keyword passes, its return value is returned.

All normal failures are caught by this keyword. Errors caused by invalid syntax, test or keyword timeouts, or fatal exceptions (caused e.g. by *Fatal Error*) are not caught.

Running the same keyword multiple times inside this keyword can create lots of output and considerably increase the size of the generated output files. It is possible to remove unnecessary keywords from the outputs using `--RemoveKeywords WUKS` command line option.

exception robot.libraries.BuiltIn.**RobotNotRunningError**
Bases: exceptions.AttributeError

Used when something cannot be done because Robot is not running.

Based on AttributeError to be backwards compatible with RF < 2.8.5. May later be based directly on Exception, so new code should except this exception explicitly.

args

message

robot.libraries.BuiltIn.**register_run_keyword**(*library*, *keyword*, *args_to_process=None*, *deprecation_warning=True*)
Tell Robot Framework that this keyword runs other keywords internally.

NOTE: This API will change in the future. For more information see https://github.com/robotframework/robotframework/issues/2190. Use with *deprecation_warning=False* to avoid related deprecation warnings.

1) Why is this method needed

Keywords running other keywords internally using *Run Keyword* or its variants like *Run Keyword If* need some special handling by the framework. This includes not processing arguments (e.g. variables in them) twice, special handling of timeouts, and so on.

2) How to use this method

library is the name of the library where the registered keyword is implemented.

keyword is the name of the keyword. With Python 2 it is possible to pass also the function or method implementing the keyword.

args_to_process' defines how many of the arguments to the registered keyword must be processed normally.

3) Examples

from robot.libraries.BuiltIn import BuiltIn, register_run_keyword

def my_run_keyword(name, *args): # do something return BuiltIn().run_keyword(name, *args)

register_run_keyword(__name__, 'My Run Keyword', 1)

from robot.libraries.BuiltIn import BuiltIn, register_run_keyword

class MyLibrary:

def my_run_keyword_if(self, expression, name, *args): # do something return BuiltIn().run_keyword_if(expression, name, *args)

register_run_keyword('MyLibrary', 'my_run_keyword_if', 2)

robot.libraries.Collections module

class robot.libraries.Collections.**NotSet**
Bases: object

class robot.libraries.Collections.**Collections**
Bases: robot.libraries.Collections._List, robot.libraries.Collections._Dictionary

A test library providing keywords for handling lists and dictionaries.

Collections is Robot Framework's standard library that provides a set of keywords for handling Python lists and dictionaries. This library has keywords, for example, for modifying and getting values from lists and

dictionaries (e.g. *Append To List*, *Get From Dictionary*) and for verifying their contents (e.g. *Lists Should Be Equal*, *Dictionary Should Contain Value*).

== Table of contents ==

%TOC%

= Related keywords in BuiltIn =

Following keywords in the BuiltIn library can also be used with lists and dictionaries:

= Using with list-like and dictionary-like objects =

List keywords that do not alter the given list can also be used with tuples, and to some extend also with other iterables. *Convert To List* can be used to convert tuples and other iterables to Python `list` objects.

Similarly dictionary keywords can, for most parts, be used with other mappings. *Convert To Dictionary* can be used if real Python `dict` objects are needed.

= Boolean arguments =

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-insensitively. Keywords verifying something that allow dropping actual and expected values from the possible error message also consider string `no values` to be false. Other strings are considered true regardless their value, and other argument types are tested using the same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

True examples:

False examples:

Considering `OFF` and `0` false is new in Robot Framework 3.1.

= Data in examples =

List related keywords use variables in format `${Lx}` in their examples. They mean lists with as many alphabetic characters as specified by `x`. For example, `${L1}` means `['a']` and `${L3}` means `['a', 'b', 'c']`.

Dictionary keywords use similar `${Dx}` variables. For example, `${D1}` means `{'a': 1}` and `${D3}` means `{'a': 1, 'b': 2, 'c': 3}`.

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

ROBOT_LIBRARY_VERSION = '4.0.3'

should_contain_match(*list*, *pattern*, *msg=None*, *case_insensitive=False*, *whitespace_insensitive=False*)

Fails if `pattern` is not found in `list`.

By default, pattern matching is similar to matching files in a shell and is case-sensitive and whitespace-sensitive. In the pattern syntax, `*` matches to anything and `?` matches to any single character. You can also prepend `glob=` to your pattern to explicitly use this pattern matching behavior.

If you prepend `regexp=` to your pattern, your pattern will be used according to the Python [http://docs.python.org/library/re.html|re module] regular expression syntax. Important note: Backslashes are an escape character, and must be escaped with another backslash (e.g. `regexp=\\d{6}` to search for `\d{6}`). See *BuiltIn.Should Match Regexp* for more details.

If `case_insensitive` is given a true value (see *Boolean arguments*), the pattern matching will ignore case.

If `whitespace_insensitive` is given a true value (see *Boolean arguments*), the pattern matching will ignore whitespace.

Non-string values in lists are ignored when matching patterns.

Use the `msg` argument to override the default error message.

See also `Should Not Contain Match`.

should_not_contain_match(*list*, *pattern*, *msg=None*, *case_insensitive=False*, *whitespace_insensitive=False*)
Fails if `pattern` is found in `list`.

Exact opposite of *Should Contain Match* keyword. See that keyword for information about arguments and usage in general.

get_matches(*list*, *pattern*, *case_insensitive=False*, *whitespace_insensitive=False*)
Returns a list of matches to `pattern` in `list`.

For more information on `pattern`, `case_insensitive`, and `whitespace_insensitive`, see *Should Contain Match*.

get_match_count(*list*, *pattern*, *case_insensitive=False*, *whitespace_insensitive=False*)
Returns the count of matches to `pattern` in `list`.

For more information on `pattern`, `case_insensitive`, and `whitespace_insensitive`, see *Should Contain Match*.

append_to_list(*list_*, **values*)
Adds `values` to the end of `list`.

combine_lists(**lists*)
Combines the given `lists` together and returns the result.

The given lists are not altered by this keyword.

convert_to_dictionary(*item*)
Converts the given `item` to a Python `dict` type.

Mainly useful for converting other mappings to normal dictionaries. This includes converting Robot Framework's own `DotDict` instances that it uses if variables are created using the `&{var}` syntax.

Use *Create Dictionary* from the BuiltIn library for constructing new dictionaries.

convert_to_list(*item*)
Converts the given `item` to a Python `list` type.

Mainly useful for converting tuples and other iterable to lists. Use *Create List* from the BuiltIn library for constructing new lists.

copy_dictionary(*dictionary*, *deepcopy=False*)
Returns a copy of the given dictionary.

The `deepcopy` argument controls should the returned dictionary be a [https://docs.python.org/library/copy.html|shallow or deep copy]. By default returns a shallow copy, but that can be changed by giving `deepcopy` a true value (see *Boolean arguments*). This is a new option in Robot Framework 3.1.2. Earlier versions always returned shallow copies.

The given dictionary is never altered by this keyword.

copy_list(*list_*, *deepcopy=False*)
Returns a copy of the given list.

If the optional `deepcopy` is given a true value, the returned list is a deep copy. New option in Robot Framework 3.1.2.

The given list is never altered by this keyword.

count_values_in_list(*list_*, *value*, *start=0*, *end=None*)
Returns the number of occurrences of the given `value` in `list`.

The search can be narrowed to the selected sublist by the `start` and `end` indexes having the same semantics as with *Get Slice From List* keyword. The given list is never altered by this keyword.

dictionaries_should_be_equal (*dict1*, *dict2*, *msg=None*, *values=True*)
Fails if the given dictionaries are not equal.

First the equality of dictionaries' keys is checked and after that all the key value pairs. If there are differences between the values, those are listed in the error message. The types of the dictionaries do not need to be same.

See *Lists Should Be Equal* for more information about configuring the error message with `msg` and `values` arguments.

dictionary_should_contain_item (*dictionary*, *key*, *value*, *msg=None*)
An item of `key` / `value` must be found in a `dictionary`.

Value is converted to unicode for comparison.

Use the `msg` argument to override the default error message.

dictionary_should_contain_key (*dictionary*, *key*, *msg=None*)
Fails if `key` is not found from `dictionary`.

Use the `msg` argument to override the default error message.

dictionary_should_contain_sub_dictionary (*dict1*, *dict2*, *msg=None*, *values=True*)
Fails unless all items in `dict2` are found from `dict1`.

See *Lists Should Be Equal* for more information about configuring the error message with `msg` and `values` arguments.

dictionary_should_contain_value (*dictionary*, *value*, *msg=None*)
Fails if `value` is not found from `dictionary`.

Use the `msg` argument to override the default error message.

dictionary_should_not_contain_key (*dictionary*, *key*, *msg=None*)
Fails if `key` is found from `dictionary`.

Use the `msg` argument to override the default error message.

dictionary_should_not_contain_value (*dictionary*, *value*, *msg=None*)
Fails if `value` is found from `dictionary`.

Use the `msg` argument to override the default error message.

get_dictionary_items (*dictionary*, *sort_keys=True*)
Returns items of the given `dictionary` as a list.

Uses *Get Dictionary Keys* to get keys and then returns corresponding items. By default keys are sorted and items returned in that order, but this can be changed by giving `sort_keys` a false value (see *Boolean arguments*). Notice that with Python 3.5 and earlier dictionary order is undefined unless using ordered dictionaries.

Items are returned as a flat list so that first item is a key, second item is a corresponding value, third item is the second key, and so on.

The given `dictionary` is never altered by this keyword.

`sort_keys` is a new option in Robot Framework 3.1.2. Earlier items were always sorted based on keys.

get_dictionary_keys (*dictionary*, *sort_keys=True*)
Returns keys of the given `dictionary` as a list.

By default keys are returned in sorted order (assuming they are sortable), but they can be returned in the original order by giving `sort_keys` a false value (see *Boolean arguments*). Notice that with Python 3.5 and earlier dictionary order is undefined unless using ordered dictionaries.

The given `dictionary` is never altered by this keyword.

`sort_keys` is a new option in Robot Framework 3.1.2. Earlier keys were always sorted.

get_dictionary_values (*dictionary*, *sort_keys=True*)
Returns values of the given `dictionary` as a list.

Uses *Get Dictionary Keys* to get keys and then returns corresponding values. By default keys are sorted and values returned in that order, but this can be changed by giving `sort_keys` a false value (see *Boolean arguments*). Notice that with Python 3.5 and earlier dictionary order is undefined unless using ordered dictionaries.

The given `dictionary` is never altered by this keyword.

`sort_keys` is a new option in Robot Framework 3.1.2. Earlier values were always sorted based on keys.

get_from_dictionary (*dictionary*, *key*)
Returns a value from the given `dictionary` based on the given `key`.

If the given `key` cannot be found from the `dictionary`, this keyword fails.

The given dictionary is never altered by this keyword.

get_from_list (*list_*, *index*)
Returns the value specified with an `index` from `list`.

The given list is never altered by this keyword.

Index `0` means the first position, `1` the second, and so on. Similarly, `-1` is the last position, `-2` the second last, and so on. Using an index that does not exist on the list causes an error. The index can be either an integer or a string that can be converted to an integer.

get_index_from_list (*list_*, *value*, *start=0*, *end=None*)
Returns the index of the first occurrence of the `value` on the list.

The search can be narrowed to the selected sublist by the `start` and `end` indexes having the same semantics as with *Get Slice From List* keyword. In case the value is not found, -1 is returned. The given list is never altered by this keyword.

get_slice_from_list (*list_*, *start=0*, *end=None*)
Returns a slice of the given list between `start` and `end` indexes.

The given list is never altered by this keyword.

If both `start` and `end` are given, a sublist containing values from `start` to `end` is returned. This is the same as `list[start:end]` in Python. To get all items from the beginning, use 0 as the start value, and to get all items until and including the end, use `None` (default) as the end value.

Using `start` or `end` not found on the list is the same as using the largest (or smallest) available index.

insert_into_list (*list_*, *index*, *value*)
Inserts `value` into `list` to the position specified with `index`.

Index `0` adds the value into the first position, `1` to the second, and so on. Inserting from right works with negative indices so that `-1` is the second last position, `-2` third last, and so on. Use *Append To List* to add items to the end of the list.

If the absolute value of the index is greater than the length of the list, the value is added at the end (positive index) or the beginning (negative index). An index can be given either as an integer or a string that can be converted to an integer.

keep_in_dictionary (*dictionary*, **keys*)
Keeps the given `keys` in the `dictionary` and removes all other.

If the given `key` cannot be found from the `dictionary`, it is ignored.

list_should_contain_sub_list (*list1*, *list2*, *msg=None*, *values=True*)
Fails if not all of the elements in `list2` are found in `list1`.

The order of values and the number of values are not taken into account.

See *Lists Should Be Equal* for more information about configuring the error message with `msg` and `values` arguments.

list_should_contain_value (*list_*, *value*, *msg=None*)
Fails if the `value` is not found from `list`.

Use the `msg` argument to override the default error message.

list_should_not_contain_duplicates (*list_*, *msg=None*)
Fails if any element in the `list` is found from it more than once.

The default error message lists all the elements that were found from the `list` multiple times, but it can be overridden by giving a custom `msg`. All multiple times found items and their counts are also logged.

This keyword works with all iterables that can be converted to a list. The original iterable is never altered.

list_should_not_contain_value (*list_*, *value*, *msg=None*)
Fails if the `value` is found from `list`.

Use the `msg` argument to override the default error message.

lists_should_be_equal (*list1*, *list2*, *msg=None*, *values=True*, *names=None*, *ignore_order=False*)
Fails if given lists are unequal.

The keyword first verifies that the lists have equal lengths, and then it checks are all their values equal. Possible differences between the values are listed in the default error message like `Index 4: ABC != Abc`. The types of the lists do not need to be the same. For example, Python tuple and list with same content are considered equal.

The error message can be configured using `msg` and `values` arguments: - If `msg` is not given, the default error message is used. - If `msg` is given and `values` gets a value considered true

(see *Boolean arguments*), the error message starts with the given `msg` followed by a newline and the default message.

- If `msg` is given and `values` is not given a true value, the error message is just the given `msg`.

The optional `names` argument can be used for naming the indices shown in the default error message. It can either be a list of names matching the indices in the lists or a dictionary where keys are indices that need to be named. It is not necessary to name all of the indices. When using a dictionary, keys can be either integers or strings that can be converted to integers.

If the items in index 2 would differ in the above examples, the error message would contain a row like `Index 2 (email): name@foo.com != name@bar.com`.

The optional `ignore_order` argument can be used to ignore the order of the elements in the lists. Using it requires items to be sortable. This is new in Robot Framework 3.2.

log_dictionary (*dictionary*, *level='INFO'*)
Logs the size and contents of the `dictionary` using given `level`.

Valid levels are TRACE, DEBUG, INFO (default), and WARN.

If you only want to log the size, use keyword *Get Length* from the BuiltIn library.

log_list (*list_*, *level='INFO'*)
Logs the length and contents of the `list` using given `level`.

Valid levels are TRACE, DEBUG, INFO (default), and WARN.

If you only want to the length, use keyword *Get Length* from the BuiltIn library.

pop_from_dictionary (*dictionary*, *key*, *default=*)
Pops the given `key` from the `dictionary` and returns its value.

By default the keyword fails if the given `key` cannot be found from the `dictionary`. If optional `default` value is given, it will be returned instead of failing.

remove_duplicates (*list_*)
Returns a list without duplicates based on the given `list`.

Creates and returns a new list that contains all items in the given list so that one item can appear only once. Order of the items in the new list is the same as in the original except for missing duplicates. Number of the removed duplicates is logged.

remove_from_dictionary (*dictionary*, **keys*)
Removes the given `keys` from the `dictionary`.

If the given `key` cannot be found from the `dictionary`, it is ignored.

remove_from_list (*list_*, *index*)
Removes and returns the value specified with an `index` from `list`.

Index `0` means the first position, `1` the second and so on. Similarly, `-1` is the last position, `-2` the second last, and so on. Using an index that does not exist on the list causes an error. The index can be either an integer or a string that can be converted to an integer.

remove_values_from_list (*list_*, **values*)
Removes all occurrences of given `values` from `list`.

It is not an error if a value does not exist in the list at all.

reverse_list (*list_*)
Reverses the given list in place.

Note that the given list is changed and nothing is returned. Use *Copy List* first, if you need to keep also the original order.

set_list_value (*list_*, *index*, *value*)
Sets the value of `list` specified by `index` to the given `value`.

Index `0` means the first position, `1` the second and so on. Similarly, `-1` is the last position, `-2` second last, and so on. Using an index that does not exist on the list causes an error. The index can be either an integer or a string that can be converted to an integer.

set_to_dictionary (*dictionary*, **key_value_pairs*, ***items*)
Adds the given `key_value_pairs` and `items` to the `dictionary`.

Giving items as `key_value_pairs` means giving keys and values as separate arguments:

The latter syntax is typically more convenient to use, but it has a limitation that keys must be strings.

If given keys already exist in the dictionary, their values are updated.

sort_list (*list_*)
Sorts the given list in place.

Sorting fails if items in the list are not comparable with each others. On Python 2 most objects are comparable, but on Python 3 comparing, for example, strings with numbers is not possible.

Note that the given list is changed and nothing is returned. Use *Copy List* first, if you need to keep also the original order.

robot.libraries.DateTime module

A test library for handling date and time values.

`DateTime` is a Robot Framework standard library that supports creating and converting date and time values (e.g. *Get Current Date*, *Convert Time*), as well as doing simple calculations with them (e.g. *Subtract Time From Date*, *Add Time To Time*). It supports dates and times in various formats, and can also be used by other libraries programmatically.

== Table of contents ==

%TOC%

= Terminology =

In the context of this library, `date` and `time` generally have following meanings:

- **`date`: An entity with both date and time components but without any** timezone information. For example, `2014-06-11 10:07:42`.
- `time`: A time interval. For example, `1 hour 20 minutes` or `01:20:00`.

This terminology differs from what Python's standard [http://docs.python.org/library/datetime.html|datetime] module uses. Basically its [http://docs.python.org/library/datetime.html#datetime-objects|datetime] and [http://docs.python.org/library/datetime.html#timedelta-objects|timedelta] objects match `date` and `time` as defined by this library.

= Date formats =

Dates can given to and received from keywords in *timestamp*, *custom timestamp*, *Python datetime* and *epoch time* formats. These formats are discussed thoroughly in subsequent sections.

Input format is determined automatically based on the given date except when using custom timestamps, in which case it needs to be given using `date_format` argument. Default result format is timestamp, but it can be overridden using `result_format` argument.

== Timestamp ==

If a date is given as a string, it is always considered to be a timestamp. If no custom formatting is given using `date_format` argument, the timestamp is expected to be in [http://en.wikipedia.org/wiki/ISO_8601|ISO 8601] like format `YYYY-MM-DD hh:mm:ss.mil`, 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.

Dates can also be returned in the same `YYYY-MM-DD hh:mm:ss.mil` format by using `timestamp` value with `result_format` argument. This is also the default format that keywords returning dates use. Milliseconds can be excluded using `exclude_millis` as explained in *Millisecond handling* section.

== Custom timestamp ==

It is possible to use custom timestamps in both input and output. The custom format is same as accepted by Python's [http://docs.python.org/library/datetime.html#strftime-strptime-behavior| datetime.strptime] function. For example, the default timestamp discussed in the previous section would match `%Y-%m-%d %H:%M:%S.%f`.

When using a custom timestamp in input, it must be specified using `date_format` argument. The actual input value must be a string that matches the specified format exactly. When using a custom timestamp in output, it must be given using `result_format` argument.

Notice that locale aware directives like `%b` do not work correctly with Jython on non-English locales: http://bugs.jython.org/issue2285

== Python datetime ==

Python's standard [http://docs.python.org/library/datetime.html#datetime-objects|datetime] objects can be used both in input and output. In input they are recognized automatically, and in output it is possible to get them by giving `datetime` value to `result_format` argument.

One nice benefit with datetime objects is that they have different time components available as attributes that can be easily accessed using the extended variable syntax.

== Epoch time ==

Epoch time is the time in seconds since the [http://en.wikipedia.org/wiki/Unix_time|UNIX epoch] i.e. 00:00:00.000 (UTC) 1 January 1970. To give a date in epoch time, it must be given as a number (integer or float), not as a string. To return a date in epoch time, it is possible to use `epoch` value with `result_format` argument. Epoch time is returned as a floating point number.

Notice that epoch time itself is independent on timezones and thus same around the world at a certain time. What local time a certain epoch time matches obviously then depends on the timezone. For example, examples below were tested in Finland but verifications would fail on other timezones.

== Earliest supported date ==

The earliest date that is supported depends on the date format and to some extent on the platform:

- Timestamps support year 1900 and above.
- Python datetime objects support year 1 and above.
- Epoch time supports 1970 and above on Windows with Python and IronPython.
- On other platforms epoch time supports 1900 and above or even earlier.

= Time formats =

Similarly as dates, times can be given to and received from keywords in various different formats. Supported formats are *number*, *time string* (verbose and compact), *timer string* and *Python timedelta*.

Input format for time is always determined automatically based on the input. Result format is number by default, but it can be customised using `result_format` argument.

== Number ==

Time given as a number is interpreted to be seconds. It can be given either as an integer or a float, or it can be a string that can be converted to a number.

To return a time as a number, `result_format` argument must have value `number`, which is also the default. Returned number is always a float.

== Time string ==

Time strings are strings in format like `1 minute 42 seconds` or `1min 42s`. 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 a minus 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

When returning a time string, it is possible to select between verbose and compact representations using result_format argument. The verbose format uses long specifiers day, hour, minute, second and millisecond, and adds s at the end when needed. The compact format uses shorter specifiers d, h, min, s and ms, and even drops the space between the number and the specifier.

== Timer string ==

Timer string is a string 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 a minus prefix.

To return a time as timer string, result_format argument must be given value timer. Timer strings are by default returned in full hh:mm:ss.mil format, but milliseconds can be excluded using exclude_millis as explained in *Millisecond handling* section.

== Python timedelta ==

Python's standard [http://docs.python.org/library/datetime.html#datetime.timedelta|timedelta] objects are also supported both in input and in output. In input they are recognized automatically, and in output it is possible to receive them by giving timedelta value to result_format argument.

= Millisecond handling =

This library handles dates and times internally using the precision of the given input. With *timestamp*, *time string*, and *timer string* result formats seconds are, however, rounded to millisecond accuracy. Milliseconds may also be included even if there would be none.

All keywords returning dates or times have an option to leave milliseconds out by giving a true value to exclude_millis argument. If the argument is given as a string, it is considered true unless it is empty or case-insensitively equal to false, none or no. Other argument types are tested using same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

When milliseconds are excluded, seconds in returned dates and times are rounded to the nearest full second. With *timestamp* and *timer string* result formats, milliseconds will also be removed from the returned string altogether.

= Programmatic usage =

In addition to be used as normal library, this library is intended to provide a stable API for other libraries to use if they want to support same date and time formats as this library. All the provided keywords are available as functions that can be easily imported:

Additionally helper classes Date and Time can be used directly:

robot.libraries.DateTime.**get_current_date**(*time_zone='local'*, *increment=0*, *result_format='timestamp'*, *exclude_millis=False*)

Returns current local or UTC time with an optional increment.

Arguments: - time_zone: Get the current time on this time zone. Currently only

local (default) and UTC are supported.

- **increment: Optional time increment to add to the returned date in** one of the supported *time formats*. Can be negative.
- result_format: Format of the returned date (see *date formats*).
- **exclude_millis: When set to any true value, rounds and drops** milliseconds as explained in *millisecond handling*.

robot.libraries.DateTime.**convert_date**(*date*, *result_format='timestamp'*, *exclude_millis=False*, *date_format=None*)

Converts between supported *date formats*.

Arguments: - date: Date in one of the supported *date formats*. - result_format: Format of the returned date. - exclude_millis: When set to any true value, rounds and drops

milliseconds as explained in *millisecond handling*.

- date_format: Specifies possible *custom timestamp* format.

robot.libraries.DateTime.**convert_time**(*time*, *result_format='number'*, *exclude_millis=False*)

Converts between supported *time formats*.

Arguments: - time: Time in one of the supported *time formats*. - result_format: Format of the returned time. - exclude_millis: When set to any true value, rounds and drops

milliseconds as explained in *millisecond handling*.

robot.libraries.DateTime.**subtract_date_from_date**(*date1*, *date2*, *result_format='number'*, *exclude_millis=False*, *date1_format=None*, *date2_format=None*)

Subtracts date from another date and returns time between.

Arguments: - date1: Date to subtract another date from in one of the

supported *date formats*.

- **date2: Date that is subtracted in one of the supported** *date formats*.
- result_format: Format of the returned time (see *time formats*).
- **exclude_millis: When set to any true value, rounds and drops** milliseconds as explained in *millisecond handling*.
- date1_format: Possible *custom timestamp* format of date1.
- date2_format: Possible *custom timestamp* format of date2.

Examples:

robot.libraries.DateTime.**add_time_to_date**(*date*, *time*, *result_format='timestamp'*, *exclude_millis=False*, *date_format=None*)

Adds time to date and returns the resulting date.

Arguments: - date: Date to add time to in one of the supported

date formats.

- **time: Time that is added in one of the supported** *time formats*.
- result_format: Format of the returned date.

- **exclude_millis: When set to any true value, rounds and drops** milliseconds as explained in *millisecond handling*.
- date_format: Possible *custom timestamp* format of date.

robot.libraries.DateTime.**subtract_time_from_date**(*date*, *time*, *result_format='timestamp'*, *exclude_millis=False*, *date_format=None*)

Subtracts time from date and returns the resulting date.

Arguments: - date: Date to subtract time from in one of the supported

date formats.

- **time: Time that is subtracted in one of the supported** *time formats*.
- result_format: Format of the returned date.
- **exclude_millis: When set to any true value, rounds and drops** milliseconds as explained in *millisecond handling*.
- date_format: Possible *custom timestamp* format of date.

robot.libraries.DateTime.**add_time_to_time**(*time1*, *time2*, *result_format='number'*, *exclude_millis=False*)

Adds time to another time and returns the resulting time.

Arguments: - time1: First time in one of the supported *time formats*. - time2: Second time in one of the supported *time formats*. - result_format: Format of the returned time. - exclude_millis: When set to any true value, rounds and drops

milliseconds as explained in *millisecond handling*.

robot.libraries.DateTime.**subtract_time_from_time**(*time1*, *time2*, *result_format='number'*, *exclude_millis=False*)

Subtracts time from another time and returns the resulting time.

Arguments: - time1: Time to subtract another time from in one of

the supported *time formats*.

- time2: Time to subtract in one of the supported *time formats*.
- result_format: Format of the returned time.
- **exclude_millis: When set to any true value, rounds and drops** milliseconds as explained in *millisecond handling*.

robot.libraries.Dialogs module

A test library providing dialogs for interacting with users.

Dialogs is Robot Framework's standard library that provides means for pausing the test execution and getting input from users. The dialogs are slightly different depending on whether tests are run on Python, IronPython or Jython but they provide the same functionality.

Long lines in the provided messages are wrapped automatically. If you want to wrap lines manually, you can add newlines using the \n character sequence.

The library has a known limitation that it cannot be used with timeouts on Python.

robot.libraries.Dialogs.**pause_execution**(*message='Test execution paused. Press OK to continue.'*)

Pauses test execution until user clicks `Ok` button.

`message` is the message shown in the dialog.

robot.libraries.Dialogs.**execute_manual_step**(*message*, *default_error=''*)

Pauses test execution until user sets the keyword status.

User can press either `PASS` or `FAIL` button. In the latter case execution fails and an additional dialog is opened for defining the error message.

`message` is the instruction shown in the initial dialog and `default_error` is the default value shown in the possible error message dialog.

robot.libraries.Dialogs.**get_value_from_user**(*message*, *default_value=''*, *hidden=False*)

Pauses test execution and asks user to input a value.

Value typed by the user, or the possible default value, is returned. Returning an empty value is fine, but pressing `Cancel` fails the keyword.

`message` is the instruction shown in the dialog and `default_value` is the possible default value shown in the input field.

If `hidden` is given a true value, the value typed by the user is hidden. `hidden` is considered true if it is a non-empty string not equal to `false`, `none` or `no`, case-insensitively. If it is not a string, its truth value is got directly using same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

robot.libraries.Dialogs.**get_selection_from_user**(*message*, **values*)

Pauses test execution and asks user to select a value.

The selected value is returned. Pressing `Cancel` fails the keyword.

`message` is the instruction shown in the dialog and `values` are the options given to the user.

robot.libraries.Dialogs.**get_selections_from_user**(*message*, **values*)

Pauses test execution and asks user to select multiple values.

The selected values are returned as a list. Selecting no values is OK and in that case the returned list is empty. Pressing `Cancel` fails the keyword.

`message` is the instruction shown in the dialog and `values` are the options given to the user.

New in Robot Framework 3.1.

robot.libraries.Easter module

robot.libraries.Easter.**none_shall_pass**(*who*)

robot.libraries.OperatingSystem module

class robot.libraries.OperatingSystem.**OperatingSystem**

Bases: `object`

A test library providing keywords for OS related tasks.

`OperatingSystem` is Robot Framework's standard library that enables various operating system related tasks to be performed in the system where Robot Framework is running. It can, among other things, execute commands (e.g. *Run*), create and remove files and directories (e.g. *Create File*, *Remove Directory*), check whether files or directories exists or contain something (e.g. *File Should Exist*, *Directory Should Be Empty*) and manipulate environment variables (e.g. *Set Environment Variable*).

== Table of contents ==

%TOC%

= Path separators =

Because Robot Framework uses the backslash (\) as an escape character in the test data, using a literal backslash requires duplicating it like in `c:\\path\\file.txt`. That can be inconvenient especially with longer Windows paths, and thus all keywords expecting paths as arguments convert forward slashes to backslashes automatically on Windows. This also means that paths like `${CURDIR}/path/file.txt` are operating system independent.

Notice that the automatic path separator conversion does not work if the path is only a part of an argument like with *Run* and *Start Process* keywords. In these cases the built-in variable `${/}` that contains `\` or `/`, depending on the operating system, can be used instead.

= Pattern matching =

Some keywords allow their arguments to be specified as [http://en.wikipedia.org/wiki/Glob_(programming)|glob patterns] where:

Unless otherwise noted, matching is case-insensitive on case-insensitive operating systems such as Windows.

= Tilde expansion =

Paths beginning with `~` or `~username` are expanded to the current or specified user's home directory, respectively. The resulting path is operating system dependent, but typically e.g. `~/robot` is expanded to `C:\Users\<user>\robot` on Windows and `/home/<user>/robot` on Unixes.

The `~username` form does not work on Jython.

= Boolean arguments =

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-insensitively. Other strings are considered true regardless their value, and other argument types are tested using the same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

True examples:

False examples:

Considering *OFF*' and `0` false is new in Robot Framework 3.1.

= Example =

`ROBOT_LIBRARY_SCOPE = 'GLOBAL'`

`ROBOT_LIBRARY_VERSION = '4.0.3'`

`run`(*command*)

Runs the given command in the system and returns the output.

The execution status of the command *is not checked* by this keyword, and it must be done separately based on the returned output. If the execution return code is needed, either *Run And Return RC* or *Run And Return RC And Output* can be used.

The standard error stream is automatically redirected to the standard output stream by adding `2>&1` after the executed command. This automatic redirection is done only when the executed command does not contain additional output redirections. You can thus freely forward the standard error somewhere else, for example, like `my_command 2>stderr.txt`.

The returned output contains everything written into the standard output or error streams by the command (unless either of them is redirected explicitly). Many commands add an extra newline (`\n`) after the output

to make it easier to read in the console. To ease processing the returned output, this possible trailing newline is stripped by this keyword.

TIP: Run Process keyword provided by the [http://robotframework.org/robotframework/latest/libraries/Process.html| Process library] supports better process configuration and is generally recommended as a replacement for this keyword.

run_and_return_rc (*command*)
Runs the given command in the system and returns the return code.

The return code (RC) is returned as a positive integer in range from 0 to 255 as returned by the executed command. On some operating systems (notable Windows) original return codes can be something else, but this keyword always maps them to the 0-255 range. Since the RC is an integer, it must be checked e.g. with the keyword *Should Be Equal As Integers* instead of *Should Be Equal* (both are built-in keywords).

See *Run* and *Run And Return RC And Output* if you need to get the output of the executed command.

TIP: Run Process keyword provided by the [http://robotframework.org/robotframework/latest/libraries/Process.html| Process library] supports better process configuration and is generally recommended as a replacement for this keyword.

run_and_return_rc_and_output (*command*)
Runs the given command in the system and returns the RC and output.

The return code (RC) is returned similarly as with *Run And Return RC* and the output similarly as with *Run*.

TIP: Run Process keyword provided by the [http://robotframework.org/robotframework/latest/libraries/Process.html| Process library] supports better process configuration and is generally recommended as a replacement for this keyword.

get_file (*path*, *encoding='UTF-8'*, *encoding_errors='strict'*)
Returns the contents of a specified file.

This keyword reads the specified file and returns the contents. Line breaks in content are converted to platform independent form. See also *Get Binary File*.

`encoding` defines the encoding of the file. The default value is `UTF-8`, which means that UTF-8 and ASCII encoded files are read correctly. In addition to the encodings supported by the underlying Python implementation, the following special encoding values can be used:

- `SYSTEM`: Use the default system encoding.
- `CONSOLE`: Use the console encoding. Outside Windows this is same as the system encoding.

`encoding_errors` argument controls what to do if decoding some bytes fails. All values accepted by `decode` method in Python are valid, but in practice the following values are most useful:

- `strict`: Fail if characters cannot be decoded (default).
- `ignore`: Ignore characters that cannot be decoded.
- `replace`: Replace characters that cannot be decoded with a replacement character.

get_binary_file (*path*)
Returns the contents of a specified file.

This keyword reads the specified file and returns the contents as is. See also *Get File*.

grep_file (*path*, *pattern*, *encoding='UTF-8'*, *encoding_errors='strict'*)
Returns the lines of the specified file that match the `pattern`.

This keyword reads a file from the file system using the defined `path`, `encoding` and `encoding_errors` similarly as *Get File*. A difference is that only the lines that match the given

`pattern` are returned. Lines are returned as a single string catenated back together with newlines and the number of matched lines is automatically logged. Possible trailing newline is never returned.

A line matches if it contains the `pattern` anywhere in it and it *does not need to match the pattern fully*. The pattern matching syntax is explained in *introduction*, and in this case matching is case-sensitive.

If more complex pattern matching is needed, it is possible to use *Get File* in combination with String library keywords like *Get Lines Matching Regexp*.

This keyword supports special `SYSTEM` and `CONSOLE` encodings that *Get File* supports only with Robot Framework 4.0 and newer. When using Python 3, it is possible to use `${NONE}` instead of `SYSTEM` with earlier versions.

log_file(*path*, *encoding='UTF-8'*, *encoding_errors='strict'*)
Wrapper for *Get File* that also logs the returned file.

The file is logged with the INFO level. If you want something else, just use *Get File* and the built-in keyword *Log* with the desired level.

See *Get File* for more information about `encoding` and `encoding_errors` arguments.

should_exist(*path*, *msg=None*)
Fails unless the given path (file or directory) exists.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

should_not_exist(*path*, *msg=None*)
Fails if the given path (file or directory) exists.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

file_should_exist(*path*, *msg=None*)
Fails unless the given `path` points to an existing file.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

file_should_not_exist(*path*, *msg=None*)
Fails if the given path points to an existing file.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

directory_should_exist(*path*, *msg=None*)
Fails unless the given path points to an existing directory.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

directory_should_not_exist(*path*, *msg=None*)
Fails if the given path points to an existing file.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. The default error message can be overridden with the `msg` argument.

wait_until_removed(*path*, *timeout='1 minute'*)
Waits until the given file or directory is removed.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. If the path is a pattern, the keyword waits until all matching items are removed.

The optional `timeout` can be used to control the maximum time of waiting. The timeout is given as a timeout string, e.g. in a format `15 seconds`, `1min 10s` or just `10`. The time string format is described in an appendix of Robot Framework User Guide.

If the timeout is negative, the keyword is never timed-out. The keyword returns immediately, if the path does not exist in the first place.

wait_until_created(*path*, *timeout='1 minute'*)
Waits until the given file or directory is created.

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. If the path is a pattern, the keyword returns when an item matching it is created.

The optional `timeout` can be used to control the maximum time of waiting. The timeout is given as a timeout string, e.g. in a format `15 seconds`, `1min 10s` or just `10`. The time string format is described in an appendix of Robot Framework User Guide.

If the timeout is negative, the keyword is never timed-out. The keyword returns immediately, if the path already exists.

directory_should_be_empty(*path*, *msg=None*)
Fails unless the specified directory is empty.

The default error message can be overridden with the `msg` argument.

directory_should_not_be_empty(*path*, *msg=None*)
Fails if the specified directory is empty.

The default error message can be overridden with the `msg` argument.

file_should_be_empty(*path*, *msg=None*)
Fails unless the specified file is empty.

The default error message can be overridden with the `msg` argument.

file_should_not_be_empty(*path*, *msg=None*)
Fails if the specified directory is empty.

The default error message can be overridden with the `msg` argument.

create_file(*path*, *content=''*, *encoding='UTF-8'*)
Creates a file with the given content and encoding.

If the directory where the file is created does not exist, it is automatically created along with possible missing intermediate directories. Possible existing file is overwritten.

On Windows newline characters (`\n`) in content are automatically converted to Windows native newline sequence (`\r\n`).

See *Get File* for more information about possible `encoding` values, including special values `SYSTEM` and `CONSOLE`.

Use *Append To File* if you want to append to an existing file and *Create Binary File* if you need to write bytes without encoding. *File Should Not Exist* can be used to avoid overwriting existing files.

Automatically converting `\n` to `\r\n` on Windows is new in Robot Framework 3.1.

create_binary_file(*path*, *content*)
Creates a binary file with the given content.

If content is given as a Unicode string, it is first converted to bytes character by character. All characters with ordinal below 256 can be used and are converted to bytes with same values. Using characters with higher ordinal is an error.

Byte strings, and possible other types, are written to the file as is.

If the directory for the file does not exist, it is created, along with missing intermediate directories.

Use *Create File* if you want to create a text file using a certain encoding. *File Should Not Exist* can be used to avoid overwriting existing files.

append_to_file (*path*, *content*, *encoding='UTF-8'*)
Appends the given content to the specified file.

If the file exists, the given text is written to its end. If the file does not exist, it is created.

Other than not overwriting possible existing files, this keyword works exactly like *Create File*. See its documentation for more details about the usage.

Note that special encodings `SYSTEM` and `CONSOLE` only work with this keyword starting from Robot Framework 3.1.2.

remove_file (*path*)
Removes a file with the given path.

Passes if the file does not exist, but fails if the path does not point to a regular file (e.g. it points to a directory).

The path can be given as an exact path or as a glob pattern. The pattern matching syntax is explained in *introduction*. If the path is a pattern, all files matching it are removed.

remove_files (**paths*)
Uses *Remove File* to remove multiple files one-by-one.

empty_directory (*path*)
Deletes all the content from the given directory.

Deletes both files and sub-directories, but the specified directory itself if not removed. Use *Remove Directory* if you want to remove the whole directory.

create_directory (*path*)
Creates the specified directory.

Also possible intermediate directories are created. Passes if the directory already exists, but fails if the path exists and is not a directory.

remove_directory (*path*, *recursive=False*)
Removes the directory pointed to by the given `path`.

If the second argument `recursive` is given a true value (see *Boolean arguments*), the directory is removed recursively. Otherwise removing fails if the directory is not empty.

If the directory pointed to by the `path` does not exist, the keyword passes, but it fails, if the `path` points to a file.

copy_file (*source*, *destination*)
Copies the source file into the destination.

Source must be a path to an existing file or a glob pattern (see *Pattern matching*) that matches exactly one file. How the destination is interpreted is explained below.

1) If the destination is an existing file, the source file is copied over it.

2) If the destination is an existing directory, the source file is copied into it. A possible file with the same name as the source is overwritten.

3) If the destination does not exist and it ends with a path separator (`/` or `\`), it is considered a directory. That directory is created and a source file copied into it. Possible missing intermediate directories are also created.

4) If the destination does not exist and it does not end with a path separator, it is considered a file. If the path to the file does not exist, it is created.

The resulting destination path is returned.

See also *Copy Files*, *Move File*, and *Move Files*.

move_file(*source*, *destination*)
Moves the source file into the destination.

Arguments have exactly same semantics as with *Copy File* keyword. Destination file path is returned.

If the source and destination are on the same filesystem, rename operation is used. Otherwise file is copied to the destination filesystem and then removed from the original filesystem.

See also *Move Files*, *Copy File*, and *Copy Files*.

copy_files(**sources_and_destination*)
Copies specified files to the target directory.

Source files can be given as exact paths and as glob patterns (see *Pattern matching*). At least one source must be given, but it is not an error if it is a pattern that does not match anything.

Last argument must be the destination directory. If the destination does not exist, it will be created.

See also *Copy File*, *Move File*, and *Move Files*.

move_files(**sources_and_destination*)
Moves specified files to the target directory.

Arguments have exactly same semantics as with *Copy Files* keyword.

See also *Move File*, *Copy File*, and *Copy Files*.

copy_directory(*source*, *destination*)
Copies the source directory into the destination.

If the destination exists, the source is copied under it. Otherwise the destination directory and the possible missing intermediate directories are created.

move_directory(*source*, *destination*)
Moves the source directory into a destination.

Uses *Copy Directory* keyword internally, and `source` and `destination` arguments have exactly same semantics as with that keyword.

get_environment_variable(*name*, *default=None*)
Returns the value of an environment variable with the given name.

If no such environment variable is set, returns the default value, if given. Otherwise fails the test case.

Returned variables are automatically decoded to Unicode using the system encoding.

Note that you can also access environment variables directly using the variable syntax `%{ENV_VAR_NAME}`.

set_environment_variable(*name*, *value*)
Sets an environment variable to a specified value.

Values are converted to strings automatically. Set variables are automatically encoded using the system encoding.

append_to_environment_variable(*name*, **values*, ***config*)
Appends given `values` to environment variable `name`.

If the environment variable already exists, values are added after it, and otherwise a new environment variable is created.

Values are, by default, joined together using the operating system path separator (`;` on Windows, `:` elsewhere). This can be changed by giving a separator after the values like `separator=value`. No other configuration parameters are accepted.

remove_environment_variable(**names*)
Deletes the specified environment variable.

Does nothing if the environment variable is not set.

It is possible to remove multiple variables by passing them to this keyword as separate arguments.

environment_variable_should_be_set(*name*, *msg=None*)
Fails if the specified environment variable is not set.

The default error message can be overridden with the `msg` argument.

environment_variable_should_not_be_set(*name*, *msg=None*)
Fails if the specified environment variable is set.

The default error message can be overridden with the `msg` argument.

get_environment_variables()
Returns currently available environment variables as a dictionary.

Both keys and values are decoded to Unicode using the system encoding. Altering the returned dictionary has no effect on the actual environment variables.

log_environment_variables(*level='INFO'*)
Logs all environment variables using the given log level.

Environment variables are also returned the same way as with *Get Environment Variables* keyword.

join_path(*base*, **parts*)
Joins the given path part(s) to the given base path.

The path separator (`/` or `\`) is inserted when needed and the possible absolute paths handled as expected. The resulted path is also normalized.

- ${path} = 'my/path'
- ${p2} = 'my/path'
- ${p3} = 'my/path/my/file.txt'
- ${p4} = '/path'
- ${p5} = '/my/path2'

join_paths(*base*, **paths*)
Joins given paths with base and returns resulted paths.

See *Join Path* for more information.

- @{p1} = ['base/example', 'base/other']
- @{p2} = ['/example', '/my/base/other']
- @{p3} = ['my/base/example/path', 'my/base/other', 'my/base/one/more']

normalize_path(*path*, *case_normalize=False*)
Normalizes the given path.

- Collapses redundant separators and up-level references.

- Converts / to \ on Windows.
- Replaces initial ~ or ~user by that user's home directory. The latter is not supported on Jython.
- If case_normalize is given a true value (see *Boolean arguments*) on Windows, converts the path to all lowercase. New in Robot Framework 3.1.
- ${path1} = 'abc'
- ${path2} = 'def'
- ${path3} = 'abc/def/ghi'
- ${path4} = '/home/robot/stuff'

On Windows result would use \ instead of / and home directory would be different.

split_path (*path*)
Splits the given path from the last path separator (/ or \).

The given path is first normalized (e.g. a possible trailing path separator is removed, special directories .. and . removed). The parts that are split are returned as separate components.

- ${path1} = 'abc' & ${dir} = 'def'
- ${path2} = 'abc/def' & ${file} = 'ghi.txt'
- ${path3} = 'def' & ${d2} = 'ghi'

split_extension (*path*)
Splits the extension from the given path.

The given path is first normalized (e.g. possible trailing path separators removed, special directories .. and . removed). The base path and extension are returned as separate components so that the dot used as an extension separator is removed. If the path contains no extension, an empty string is returned for it. Possible leading and trailing dots in the file name are never considered to be extension separators.

- ${path} = 'file' & ${ext} = 'extension'
- ${p2} = 'path/file' & ${e2} = 'ext'
- ${p3} = 'path/file' & ${e3} = ''
- ${p4} = 'p2/file' & ${e4} = 'ext'
- ${p5} = 'path/.file' & ${e5} = 'ext'
- ${p6} = 'path/.file' & ${e6} = ''

get_modified_time (*path*, *format='timestamp'*)
Returns the last modification time of a file or directory.

How time is returned is determined based on the given format string as follows. Note that all checks are case-insensitive. Returned time is also automatically logged.

1) If format contains the word epoch, the time is returned in seconds after the UNIX epoch. The return value is always an integer.
2) If format contains any of the words year, month, day, hour, min or sec, only the selected parts are returned. The order of the returned parts is always the one in the previous sentence and the order of the words in format is not significant. The parts are returned as zero-padded strings (e.g. May -> 05).
3) Otherwise, and by default, the time is returned as a timestamp string in the format 2006-02-24 15:08:31.

2006-03-29 15:06:21): - ${time} = '2006-03-29 15:06:21' - ${secs} = 1143637581 - ${year} = '2006' - ${y} = '2006' & ${d} = '29' - @{time} = ['2006', '03', '29', '15', '06', '21']

set_modified_time(*path*, *mtime*)
Sets the file modification and access times.

Changes the modification and access times of the given file to the value determined by `mtime`. The time can be given in different formats described below. Note that all checks involving strings are case-insensitive. Modified time can only be set to regular files.

1) If `mtime` is a number, or a string that can be converted to a number, it is interpreted as seconds since the UNIX epoch (1970-01-01 00:00:00 UTC). This documentation was originally written about 1177654467 seconds after the epoch.
2) If `mtime` is a timestamp, that time will be used. Valid timestamp formats are `YYYY-MM-DD hh:mm:ss` and `YYYYMMDD hhmmss`.
3) If `mtime` is equal to `NOW`, the current local time is used.
4) If `mtime` is equal to `UTC`, the current time in [http://en.wikipedia.org/wiki/Coordinated_Universal_Time|UTC] is used.
5) If `mtime` is in the format like `NOW - 1 day` or `UTC + 1 hour 30 min`, the current local/UTC time plus/minus the time specified with the time string is used. The time string format is described in an appendix of Robot Framework User Guide.

get_file_size(*path*)
Returns and logs file size as an integer in bytes.

list_directory(*path*, *pattern=None*, *absolute=False*)
Returns and logs items in a directory, optionally filtered with `pattern`.

File and directory names are returned in case-sensitive alphabetical order, e.g. `['A Name', 'Second', 'a lower case name', 'one more']`. Implicit directories `.` and `..` are not returned. The returned items are automatically logged.

File and directory names are returned relative to the given path (e.g. `'file.txt'`) by default. If you want them be returned in absolute format (e.g. `'/home/robot/file.txt'`), give the `absolute` argument a true value (see *Boolean arguments*).

If `pattern` is given, only items matching it are returned. The pattern matching syntax is explained in *introduction*, and in this case matching is case-sensitive.

list_files_in_directory(*path*, *pattern=None*, *absolute=False*)
Wrapper for *List Directory* that returns only files.

list_directories_in_directory(*path*, *pattern=None*, *absolute=False*)
Wrapper for *List Directory* that returns only directories.

count_items_in_directory(*path*, *pattern=None*)
Returns and logs the number of all items in the given directory.

The argument `pattern` has the same semantics as with *List Directory* keyword. The count is returned as an integer, so it must be checked e.g. with the built-in keyword *Should Be Equal As Integers*.

count_files_in_directory(*path*, *pattern=None*)
Wrapper for *Count Items In Directory* returning only file count.

count_directories_in_directory(*path*, *pattern=None*)
Wrapper for *Count Items In Directory* returning only directory count.

touch(*path*)
Emulates the UNIX touch command.

Creates a file, if it does not exist. Otherwise changes its access and modification times to the current time.

Fails if used with the directories or the parent directory of the given file does not exist.

robot.libraries.Process module

class robot.libraries.Process.**Process**

Bases: object

Robot Framework test library for running processes.

This library utilizes Python's [http://docs.python.org/library/subprocess.html|subprocess] module and its [http://docs.python.org/library/subprocess.html#popen-constructor|Popen] class.

The library has following main usages:

- Running processes in system and waiting for their completion using *Run Process* keyword.
- Starting processes on background using *Start Process*.
- Waiting started process to complete using *Wait For Process* or stopping them with *Terminate Process* or *Terminate All Processes*.

== Table of contents ==

%TOC%

= Specifying command and arguments =

Both *Run Process* and *Start Process* accept the command to execute and all arguments passed to the command as separate arguments. This makes usage convenient and also allows these keywords to automatically escape possible spaces and other special characters in commands and arguments. Notice that if a command accepts options that themselves accept values, these options and their values must be given as separate arguments.

When *running processes in shell*, it is also possible to give the whole command to execute as a single string. The command can then contain multiple commands to be run together. When using this approach, the caller is responsible on escaping.

Possible non-string arguments are converted to strings automatically.

= Process configuration =

Run Process and *Start Process* keywords can be configured using optional `**configuration` keyword arguments. Configuration arguments must be given after other arguments passed to these keywords and must use syntax like `name=value`. Available configuration arguments are listed below and discussed further in sections afterwards.

Note that because `**configuration` is passed using `name=value` syntax, possible equal signs in other arguments passed to *Run Process* and *Start Process* must be escaped with a backslash like `name\=value`. See *Run Process* for an example.

== Running processes in shell ==

The `shell` argument specifies whether to run the process in a shell or not. By default shell is not used, which means that shell specific commands, like `copy` and `dir` on Windows, are not available. You can, however, run shell scripts and batch files without using a shell.

Giving the `shell` argument any non-false value, such as `shell=True`, changes the program to be executed in a shell. It allows using the shell capabilities, but can also make the process invocation operating system dependent. Having a shell between the actually started process and this library can also interfere communication with the process such as stopping it and reading its outputs. Because of these problems, it is recommended to use the shell only when absolutely necessary.

When using a shell it is possible to give the whole command to execute as a single string. See *Specifying command and arguments* section for examples and more details in general.

== Current working directory ==

By default the child process will be executed in the same directory as the parent process, the process running tests, is executed. This can be changed by giving an alternative location using the `cwd` argument. Forward slashes in the given path are automatically converted to backslashes on Windows.

Standard output and error streams, when redirected to files, are also relative to the current working directory possibly set using the `cwd` argument.

== Environment variables ==

By default the child process will get a copy of the parent process's environment variables. The `env` argument can be used to give the child a custom environment as a Python dictionary. If there is a need to specify only certain environment variable, it is possible to use the `env:<name>=<value>` format to set or override only that named variables. It is also possible to use these two approaches together.

== Standard output and error streams ==

By default processes are run so that their standard output and standard error streams are kept in the memory. This works fine normally, but if there is a lot of output, the output buffers may get full and the program can hang. Additionally on Jython, everything written to these in-memory buffers can be lost if the process is terminated.

To avoid the above mentioned problems, it is possible to use `stdout` and `stderr` arguments to specify files on the file system where to redirect the outputs. This can also be useful if other processes or other keywords need to read or manipulate the outputs somehow.

Given `stdout` and `stderr` paths are relative to the *current working directory*. Forward slashes in the given paths are automatically converted to backslashes on Windows.

As a special feature, it is possible to redirect the standard error to the standard output by using `stderr=STDOUT`.

Regardless are outputs redirected to files or not, they are accessible through the *result object* returned when the process ends. Commands are expected to write outputs using the console encoding, but *output encoding* can be configured using the `output_encoding` argument if needed.

If you are not interested in outputs at all, you can explicitly ignore them by using a special value `DEVNULL` both with `stdout` and `stderr`. For example, `stdout=DEVNULL` is the same as redirecting output on console with `> /dev/null` on UNIX-like operating systems or `> NUL` on Windows. This way the process will not hang even if there would be a lot of output, but naturally output is not available after execution either.

Support for the special value `DEVNULL` is new in Robot Framework 3.2.

Note that the created output files are not automatically removed after the test run. The user is responsible to remove them if needed.

== Output encoding ==

Executed commands are, by default, expected to write outputs to the *standard output and error streams* using the encoding used by the system console. If the command uses some other encoding, that can be configured using the `output_encoding` argument. This is especially useful on Windows where the console uses a different encoding than rest of the system, and many commands use the general system encoding instead of the console encoding.

The value used with the `output_encoding` argument must be a valid encoding and must match the encoding actually used by the command. As a convenience, it is possible to use strings `CONSOLE` and `SYSTEM` to specify that the console or system encoding is used, respectively. If produced outputs use different encoding then configured, values got through the *result object* will be invalid.

== Alias ==

A custom name given to the process that can be used when selecting the *active process*.

= Active process =

The test library keeps record which of the started processes is currently active. By default it is latest process started with *Start Process*, but *Switch Process* can be used to select a different one. Using *Run Process* does not affect the active process.

The keywords that operate on started processes will use the active process by default, but it is possible to explicitly select a different process using the `handle` argument. The handle can be the identifier returned by *Start Process* or an `alias` explicitly given to *Start Process* or *Run Process*.

= Result object =

Run Process, *Wait For Process* and *Terminate Process* keywords return a result object that contains information about the process execution as its attributes. The same result object, or some of its attributes, can also be get using *Get Process Result* keyword. Attributes available in the object are documented in the table below.

= Boolean arguments =

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-insensitively. Other strings are considered true regardless their value, and other argument types are tested using the same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

True examples:

False examples:

Considering `OFF` and `0` false is new in Robot Framework 3.1.

= Example =

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

ROBOT_LIBRARY_VERSION = '4.0.3'

TERMINATE_TIMEOUT = 30

KILL_TIMEOUT = 10

run_process(*command*, **arguments*, ***configuration*)

Runs a process and waits for it to complete.

`command` and `*arguments` specify the command to execute and arguments passed to it. See *Specifying command and arguments* for more details.

`**configuration` contains additional configuration related to starting processes and waiting for them to finish. See *Process configuration* for more details about configuration related to starting processes. Configuration related to waiting for processes consists of `timeout` and `on_timeout` arguments that have same semantics as with *Wait For Process* keyword. By default there is no timeout, and if timeout is defined the default action on timeout is `terminate`.

Returns a *result object* containing information about the execution.

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

This keyword does not change the *active process*.

start_process(*command*, **arguments*, ***configuration*)
Starts a new process on background.

See *Specifying command and arguments* and *Process configuration* for more information about the arguments, and *Run Process* keyword for related examples.

Makes the started process new *active process*. Returns an identifier that can be used as a handle to activate the started process if needed.

Processes are started so that they create a new process group. This allows sending signals to and terminating also possible child processes. This is not supported on Jython.

is_process_running(*handle=None*)
Checks is the process running or not.

If `handle` is not given, uses the current *active process*.

Returns `True` if the process is still running and `False` otherwise.

process_should_be_running(*handle=None*, *error_message='Process is not running.'*)
Verifies that the process is running.

If `handle` is not given, uses the current *active process*.

Fails if the process has stopped.

process_should_be_stopped(*handle=None*, *error_message='Process is running.'*)
Verifies that the process is not running.

If `handle` is not given, uses the current *active process*.

Fails if the process is still running.

wait_for_process(*handle=None*, *timeout=None*, *on_timeout='continue'*)
Waits for the process to complete or to reach the given timeout.

The process to wait for must have been started earlier with *Start Process*. If `handle` is not given, uses the current *active process*.

`timeout` defines the maximum time to wait for the process. It can be given in [http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format| various time formats] supported by Robot Framework, for example, `42`, `42 s`, or `1 minute 30 seconds`. The timeout is ignored if it is Python `None` (default), string `NONE` (case-insensitively), zero, or negative.

`on_timeout` defines what to do if the timeout occurs. Possible values and corresponding actions are explained in the table below. Notice that reaching the timeout never fails the test.

See *Terminate Process* keyword for more details how processes are terminated and killed.

If the process ends before the timeout or it is terminated or killed, this keyword returns a *result object* containing information about the execution. If the process is left running, Python `None` is returned instead.

Ignoring timeout if it is string `NONE`, zero, or negative is new in Robot Framework 3.2.

terminate_process(*handle=None*, *kill=False*)
Stops the process gracefully or forcefully.

If `handle` is not given, uses the current *active process*.

By default first tries to stop the process gracefully. If the process does not stop in 30 seconds, or `kill` argument is given a true value, (see *Boolean arguments*) kills the process forcefully. Stops also all the child processes of the originally started process.

Waits for the process to stop after terminating it. Returns a *result object* containing information about the execution similarly as *Wait For Process*.

On Unix-like machines graceful termination is done using `TERM (15)` signal and killing using `KILL (9)`. Use *Send Signal To Process* instead if you just want to send either of these signals without waiting for the process to stop.

On Windows graceful termination is done using `CTRL_BREAK_EVENT` event and killing using Win32 API function `TerminateProcess()`.

Limitations: - Graceful termination is not supported on Windows when using Jython.

Process is killed instead.

- Stopping the whole process group is not supported when using Jython.
- On Windows forceful kill only stops the main process, not possible child processes.

terminate_all_processes(*kill=False*)
Terminates all still running processes started by this library.

This keyword can be used in suite teardown or elsewhere to make sure that all processes are stopped,

By default tries to terminate processes gracefully, but can be configured to forcefully kill them immediately. See *Terminate Process* that this keyword uses internally for more details.

send_signal_to_process(*signal*, *handle=None*, *group=False*)
Sends the given `signal` to the specified process.

If `handle` is not given, uses the current *active process*.

Signal can be specified either as an integer as a signal name. In the latter case it is possible to give the name both with or without `SIG` prefix, but names are case-sensitive. For example, all the examples below send signal `INT (2)`:

This keyword is only supported on Unix-like machines, not on Windows. What signals are supported depends on the system. For a list of existing signals on your system, see the Unix man pages related to signal handling (typically `man signal` or `man 7 signal`).

By default sends the signal only to the parent process, not to possible child processes started by it. Notice that when *running processes in shell*, the shell is the parent process and it depends on the system does the shell propagate the signal to the actual started process.

To send the signal to the whole process group, `group` argument can be set to any true value (see *Boolean arguments*). This is not supported by Jython, however.

get_process_id(*handle=None*)
Returns the process ID (pid) of the process as an integer.

If `handle` is not given, uses the current *active process*.

Notice that the pid is not the same as the handle returned by *Start Process* that is used internally by this library.

get_process_object(*handle=None*)
Return the underlying `subprocess.Popen` object.

If `handle` is not given, uses the current *active process*.

get_process_result(*handle=None*, *rc=False*, *stdout=False*, *stderr=False*, *stdout_path=False*, *stderr_path=False*)
Returns the specified *result object* or some of its attributes.

The given `handle` specifies the process whose results should be returned. If no `handle` is given, results of the current *active process* are returned. In either case, the process must have been finishes before this keyword can be used. In practice this means that processes started with *Start Process* must be finished either with *Wait For Process* or *Terminate Process* before using this keyword.

If no other arguments than the optional `handle` are given, a whole *result object* is returned. If one or more of the other arguments are given any true value, only the specified attributes of the *result object* are returned. These attributes are always returned in the same order as arguments are specified in the keyword signature. See *Boolean arguments* section for more details about true and false values.

Although getting results of a previously executed process can be handy in general, the main use case for this keyword is returning results over the remote library interface. The remote interface does not support returning the whole result object, but individual attributes can be returned without problems.

switch_process(*handle*)
Makes the specified process the current *active process*.

The handle can be an identifier returned by *Start Process* or the `alias` given to it explicitly.

split_command_line(*args*, *escaping=False*)
Splits command line string into a list of arguments.

String is split from spaces, but argument surrounded in quotes may contain spaces in them. If `escaping` is given a true value, then backslash is treated as an escape character. It can escape unquoted spaces, quotes inside quotes, and so on, but it also requires using double backslashes when using Windows paths.

join_command_line(**args*)
Joins arguments into one command line string.

In resulting command line string arguments are delimited with a space, arguments containing spaces are surrounded with quotes, and possible quotes are escaped with a backslash.

If this keyword is given only one argument and that is a list like object, then the values of that list are joined instead.

class robot.libraries.Process.**ExecutionResult**(*process*, *stdout*, *stderr*, *rc=None*, *output_encoding=None*)
Bases: `object`

stdout

stderr

close_streams()

class robot.libraries.Process.**ProcessConfiguration**(*cwd=None*, *shell=False*, *stdout=None*, *stderr=None*, *output_encoding='CONSOLE'*, *alias=None*, *env=None*, ***rest*)
Bases: `object`

get_command(*command*, *arguments*)

popen_config

result_config

robot.libraries.Remote module

class robot.libraries.Remote.**Remote**(*uri='http://127.0.0.1:8270'*, *timeout=None*)
Bases: `object`

Connects to a remote server at `uri`.

Optional `timeout` can be used to specify a timeout to wait when initially connecting to the server and if a connection accidentally closes. Timeout can be given as seconds (e.g. `60`) or using Robot Framework time format (e.g. `60s`, `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.

Timeouts do not work with IronPython.

ROBOT_LIBRARY_SCOPE = 'TEST SUITE'

get_keyword_names()

get_keyword_arguments(*name*)

get_keyword_types(*name*)

get_keyword_tags(*name*)

get_keyword_documentation(*name*)

run_keyword(*name*, *args*, *kwargs*)

class robot.libraries.Remote.**ArgumentCoercer**
Bases: object

binary = <_sre.SRE_Pattern object>

non_ascii = <_sre.SRE_Pattern object>

coerce(*argument*)

class robot.libraries.Remote.**RemoteResult**(*result*)
Bases: object

class robot.libraries.Remote.**XmlRpcRemoteClient**(*uri*, *timeout=None*)
Bases: object

get_library_information()

get_keyword_names()

get_keyword_arguments(*name*)

get_keyword_types(*name*)

get_keyword_tags(*name*)

get_keyword_documentation(*name*)

run_keyword(*name*, *args*, *kwargs*)

class robot.libraries.Remote.**TimeoutHTTPTransport**(*use_datetime=0*, *timeout=None*)
Bases: xmlrpclib.Transport

make_connection(*host*)

accept_gzip_encoding = True

close()

encode_threshold = None

get_host_info(*host*)

getparser()

parse_response(*response*)

request(*host*, *handler*, *request_body*, *verbose=0*)

send_content(*connection*, *request_body*)

send_host(*connection*, *host*)

send_request(*connection*, *handler*, *request_body*)

send_user_agent(*connection*)

single_request(*host*, *handler*, *request_body*, *verbose=0*)

user_agent = 'xmlrpclib.py/1.0.1 (by www.pythonware.com)'

class robot.libraries.Remote.**TimeoutHTTPSTransport**(*use_datetime=0*, *timeout=None*)
Bases: *robot.libraries.Remote.TimeoutHTTPTransport*

accept_gzip_encoding = True

close()

encode_threshold = None

get_host_info(*host*)

getparser()

make_connection(*host*)

parse_response(*response*)

request(*host*, *handler*, *request_body*, *verbose=0*)

send_content(*connection*, *request_body*)

send_host(*connection*, *host*)

send_request(*connection*, *handler*, *request_body*)

send_user_agent(*connection*)

single_request(*host*, *handler*, *request_body*, *verbose=0*)

user_agent = 'xmlrpclib.py/1.0.1 (by www.pythonware.com)'

robot.libraries.Reserved module

class robot.libraries.Reserved.**Reserved**
Bases: object

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

robot.libraries.Screenshot module

class robot.libraries.Screenshot.**Screenshot**(*screenshot_directory=None*, *screenshot_module=None*)
Bases: object

Test library for taking screenshots on the machine where tests are run.

Notice that successfully taking screenshots requires tests to be run with a physical or virtual display.

== Table of contents ==

%TOC%

= Using with Python =

How screenshots are taken when using Python depends on the operating system. On OSX screenshots are taken using the built-in `screencapture` utility. On other operating systems you need to have one of the following

tools or Python modules installed. You can specify the tool/module to use when *importing* the library. If no tool or module is specified, the first one found will be used.

- wxPython :: http://wxpython.org :: Required also by RIDE so many Robot Framework users already have this module installed.
- PyGTK :: http://pygtk.org :: This module is available by default on most Linux distributions.
- Pillow :: http://python-pillow.github.io :: Only works on Windows. Also the original PIL package is supported.
- Scrot :: http://en.wikipedia.org/wiki/Scrot :: Not used on Windows. Install with `apt-get install scrot` or similar.

= Using with Jython and IronPython =

With Jython and IronPython this library uses APIs provided by JVM and .NET platforms, respectively. These APIs are always available and thus no external modules are needed.

= Where screenshots are saved =

By default screenshots are saved into the same directory where the Robot Framework log file is written. If no log is created, screenshots are saved into the directory where the XML output file is written.

It is possible to specify a custom location for screenshots using `screenshot_directory` argument when *importing* the library and using *Set Screenshot Directory* keyword during execution. It is also possible to save screenshots using an absolute path.

= ScreenCapLibrary =

[https://github.com/mihaiparvu/ScreenCapLibrary|ScreenCapLibrary] is an external Robot Framework library that can be used as an alternative, which additionally provides support for multiple formats, adjusting the quality, using GIFs and video capturing.

Configure where screenshots are saved.

If `screenshot_directory` is not given, screenshots are saved into same directory as the log file. The directory can also be set using *Set Screenshot Directory* keyword.

`screenshot_module` specifies the module or tool to use when using this library on Python outside OSX. Possible values are `wxPython`, `PyGTK`, `PIL` and `scrot`, case-insensitively. If no value is given, the first module/tool found is used in that order. See *Using with Python* for more information.

ROBOT_LIBRARY_SCOPE = 'TEST SUITE'

ROBOT_LIBRARY_VERSION = '4.0.3'

set_screenshot_directory(*path*)

Sets the directory where screenshots are saved.

It is possible to use `/` as a path separator in all operating systems. Path to the old directory is returned.

The directory can also be set in *importing*.

take_screenshot(*name='screenshot'*, *width='800px'*)

Takes a screenshot in JPEG format and embeds it into the log file.

Name of the file where the screenshot is stored is derived from the given `name`. If the `name` ends with extension `.jpg` or `.jpeg`, the screenshot will be stored with that exact name. Otherwise a unique name is created by adding an underscore, a running index and an extension to the `name`.

The name will be interpreted to be relative to the directory where the log file is written. It is also possible to use absolute paths. Using `/` as a path separator works in all operating systems.

`width` specifies the size of the screenshot in the log file.

The path where the screenshot is saved is returned.

take_screenshot_without_embedding(*name='screenshot'*)
Takes a screenshot and links it from the log file.

This keyword is otherwise identical to *Take Screenshot* but the saved screenshot is not embedded into the log file. The screenshot is linked so it is nevertheless easily available.

class robot.libraries.Screenshot.**ScreenshotTaker**(*module_name=None*)
Bases: object

test(*path=None*)

robot.libraries.String module

class robot.libraries.String.**String**
Bases: object

A test library for string manipulation and verification.

String is Robot Framework's standard library for manipulating strings (e.g. *Replace String Using Regexp*, *Split To Lines*) and verifying their contents (e.g. *Should Be String*).

Following keywords from BuiltIn library can also be used with strings:

- *Catenate*
- *Get Length*
- *Length Should Be*
- *Should (Not) Be Empty*
- *Should (Not) Be Equal (As Strings/Integers/Numbers)*
- *Should (Not) Match (Regexp)*
- *Should (Not) Contain*
- *Should (Not) Start With*
- *Should (Not) End With*
- *Convert To String*
- *Convert To Bytes*

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

ROBOT_LIBRARY_VERSION = '4.0.3'

convert_to_lower_case(*string*)
Converts string to lower case.

Uses Python's standard [https://docs.python.org/library/stdtypes.html#str.lower|lower()] method.

convert_to_upper_case(*string*)
Converts string to upper case.

Uses Python's standard [https://docs.python.org/library/stdtypes.html#str.upper|upper()] method.

convert_to_title_case(*string*, *exclude=None*)
Converts string to title case.

Uses the following algorithm:

- Split the string to words from whitespace characters (spaces, newlines, etc.).
- Exclude words that are not all lower case. This preserves, for example, "OK" and "iPhone".
- Exclude also words listed in the optional `exclude` argument.
- Title case the first alphabetical character of each word that has not been excluded.
- Join all words together so that original whitespace is preserved.

Explicitly excluded words can be given as a list or as a string with words separated by a comma and an optional space. Excluded words are actually considered to be regular expression patterns, so it is possible to use something like "example[.!?]?" to match the word "example" on it own and also if followed by ".", "!" or "?". See *BuiltIn.Should Match Regexp* for more information about Python regular expression syntax in general and how to use it in Robot Framework test data in particular.

The reason this keyword does not use Python's standard [https://docs.python.org/library/stdtypes.html#str.title|title()] method is that it can yield undesired results, for example, if strings contain upper case letters or special characters like apostrophes. It would, for example, convert "it's an OK iPhone" to "It'S An Ok Iphone".

New in Robot Framework 3.2.

encode_string_to_bytes(*string*, *encoding*, *errors='strict'*)
Encodes the given Unicode `string` to bytes using the given `encoding`.

`errors` argument controls what to do if encoding some characters fails. All values accepted by `encode` method in Python are valid, but in practice the following values are most useful:

- `strict`: fail if characters cannot be encoded (default)
- `ignore`: ignore characters that cannot be encoded
- `replace`: replace characters that cannot be encoded with a replacement character

Use *Convert To Bytes* in `BuiltIn` if you want to create bytes based on character or integer sequences. Use *Decode Bytes To String* if you need to convert byte strings to Unicode strings and *Convert To String* in `BuiltIn` if you need to convert arbitrary objects to Unicode.

decode_bytes_to_string(*bytes*, *encoding*, *errors='strict'*)
Decodes the given `bytes` to a Unicode string using the given `encoding`.

`errors` argument controls what to do if decoding some bytes fails. All values accepted by `decode` method in Python are valid, but in practice the following values are most useful:

- `strict`: fail if characters cannot be decoded (default)
- `ignore`: ignore characters that cannot be decoded
- `replace`: replace characters that cannot be decoded with a replacement character

Use *Encode String To Bytes* if you need to convert Unicode strings to byte strings, and *Convert To String* in `BuiltIn` if you need to convert arbitrary objects to Unicode strings.

format_string(*template*, **positional*, ***named*)
Formats a `template` using the given `positional` and `named` arguments.

The template can be either be a string or an absolute path to an existing file. In the latter case the file is read and its contents are used as the template. If the template file contains non-ASCII characters, it must be encoded using UTF-8.

The template is formatted using Python's [https://docs.python.org/library/string.html#format-string-syntax|format string syntax]. Placeholders are marked using `{}` with possible field name and format specification inside. Literal curly braces can be inserted by doubling them like *{{* and *}}*.

New in Robot Framework 3.1.

get_line_count (*string*)
Returns and logs the number of lines in the given string.

split_to_lines (*string*, *start=0*, *end=None*)
Splits the given string to lines.

It is possible to get only a selection of lines from `start` to `end` so that `start` index is inclusive and `end` is exclusive. Line numbering starts from 0, and it is possible to use negative indices to refer to lines from the end.

Lines are returned without the newlines. The number of returned lines is automatically logged.

Use *Get Line* if you only need to get a single line.

get_line (*string*, *line_number*)
Returns the specified line from the given `string`.

Line numbering starts from 0 and it is possible to use negative indices to refer to lines from the end. The line is returned without the newline character.

Use *Split To Lines* if all lines are needed.

get_lines_containing_string (*string*, *pattern*, *case_insensitive=False*)
Returns lines of the given `string` that contain the `pattern`.

The `pattern` is always considered to be a normal string, not a glob or regexp pattern. A line matches if the `pattern` is found anywhere on it.

The match is case-sensitive by default, but giving `case_insensitive` a true value makes it case-insensitive. The value is considered true if it is a non-empty string that is not equal to `false`, `none` or `no`. If the value is not a string, its truth value is got directly in Python.

Lines are returned as one string catenated back together with newlines. Possible trailing newline is never returned. The number of matching lines is automatically logged.

See *Get Lines Matching Pattern* and *Get Lines Matching Regexp* if you need more complex pattern matching.

get_lines_matching_pattern (*string*, *pattern*, *case_insensitive=False*)
Returns lines of the given `string` that match the `pattern`.

The `pattern` is a _glob **pattern_** where:

A line matches only if it matches the `pattern` fully.

The match is case-sensitive by default, but giving `case_insensitive` a true value makes it case-insensitive. The value is considered true if it is a non-empty string that is not equal to `false`, `none` or `no`. If the value is not a string, its truth value is got directly in Python.

Lines are returned as one string catenated back together with newlines. Possible trailing newline is never returned. The number of matching lines is automatically logged.

See *Get Lines Matching Regexp* if you need more complex patterns and *Get Lines Containing String* if searching literal strings is enough.

get_lines_matching_regexp (*string*, *pattern*, *partial_match=False*)
Returns lines of the given `string` that match the regexp `pattern`.

See *BuiltIn.Should Match Regexp* for more information about Python regular expression syntax in general and how to use it in Robot Framework test data in particular.

By default lines match only if they match the pattern fully, but partial matching can be enabled by giving the `partial_match` argument a true value. The value is considered true if it is a non-empty string that is not equal to `false`, `none` or `no`. If the value is not a string, its truth value is got directly in Python.

If the pattern is empty, it matches only empty lines by default. When partial matching is enabled, empty pattern matches all lines.

Notice that to make the match case-insensitive, you need to prefix the pattern with case-insensitive flag `(?i)`.

Lines are returned as one string concatenated back together with newlines. Possible trailing newline is never returned. The number of matching lines is automatically logged.

See *Get Lines Matching Pattern* and *Get Lines Containing String* if you do not need full regular expression powers (and complexity).

get_regexp_matches (*string*, *pattern*, **groups*)
Returns a list of all non-overlapping matches in the given string.

`string` is the string to find matches from and `pattern` is the regular expression. See *BuiltIn.Should Match Regexp* for more information about Python regular expression syntax in general and how to use it in Robot Framework test data in particular.

If no groups are used, the returned list contains full matches. If one group is used, the list contains only contents of that group. If multiple groups are used, the list contains tuples that contain individual group contents. All groups can be given as indexes (starting from 1) and named groups also as names.

replace_string (*string*, *search_for*, *replace_with*, *count=-1*)
Replaces `search_for` in the given `string` with `replace_with`.

`search_for` is used as a literal string. See *Replace String Using Regexp* if more powerful pattern matching is needed. If you need to just remove a string see *Remove String*.

If the optional argument `count` is given, only that many occurrences from left are replaced. Negative `count` means that all occurrences are replaced (default behaviour) and zero means that nothing is done.

A modified version of the string is returned and the original string is not altered.

replace_string_using_regexp (*string*, *pattern*, *replace_with*, *count=-1*)
Replaces `pattern` in the given `string` with `replace_with`.

This keyword is otherwise identical to *Replace String*, but the `pattern` to search for is considered to be a regular expression. See *BuiltIn.Should Match Regexp* for more information about Python regular expression syntax in general and how to use it in Robot Framework test data in particular.

If you need to just remove a string see *Remove String Using Regexp*.

remove_string (*string*, **removables*)
Removes all `removables` from the given `string`.

`removables` are used as literal strings. Each removable will be matched to a temporary string from which preceding removables have been already removed. See second example below.

Use *Remove String Using Regexp* if more powerful pattern matching is needed. If only a certain number of matches should be removed, *Replace String* or *Replace String Using Regexp* can be used.

A modified version of the string is returned and the original string is not altered.

remove_string_using_regexp (*string*, **patterns*)
Removes `patterns` from the given `string`.

This keyword is otherwise identical to *Remove String*, but the `patterns` to search for are considered to be a regular expression. See *Replace String Using Regexp* for more information about the regular expression syntax. That keyword can also be used if there is a need to remove only a certain number of occurrences.

split_string(*string*, *separator=None*, *max_split=-1*)
Splits the `string` using `separator` as a delimiter string.

If a `separator` is not given, any whitespace string is a separator. In that case also possible consecutive whitespace as well as leading and trailing whitespace is ignored.

Split words are returned as a list. If the optional `max_split` is given, at most `max_split` splits are done, and the returned list will have maximum `max_split + 1` elements.

See *Split String From Right* if you want to start splitting from right, and *Fetch From Left* and *Fetch From Right* if you only want to get first/last part of the string.

split_string_from_right(*string*, *separator=None*, *max_split=-1*)
Splits the `string` using `separator` starting from right.

Same as *Split String*, but splitting is started from right. This has an effect only when `max_split` is given.

split_string_to_characters(*string*)
Splits the given `string` to characters.

fetch_from_left(*string*, *marker*)
Returns contents of the `string` before the first occurrence of `marker`.

If the `marker` is not found, whole string is returned.

See also *Fetch From Right*, *Split String* and *Split String From Right*.

fetch_from_right(*string*, *marker*)
Returns contents of the `string` after the last occurrence of `marker`.

If the `marker` is not found, whole string is returned.

See also *Fetch From Left*, *Split String* and *Split String From Right*.

generate_random_string(*length=8*, *chars='[LETTERS][NUMBERS]'*)
Generates a string with a desired `length` from the given `chars`.

The population sequence `chars` contains the characters to use when generating the random string. It can contain any characters, and it is possible to use special markers explained in the table below:

get_substring(*string*, *start*, *end=None*)
Returns a substring from `start` index to `end` index.

The `start` index is inclusive and `end` is exclusive. Indexing starts from 0, and it is possible to use negative indices to refer to characters from the end.

strip_string(*string*, *mode='both'*, *characters=None*)
Remove leading and/or trailing whitespaces from the given string.

`mode` is either `left` to remove leading characters, `right` to remove trailing characters, `both` (default) to remove the characters from both sides of the string or `none` to return the unmodified string.

If the optional `characters` is given, it must be a string and the characters in the string will be stripped in the string. Please note, that this is not a substring to be removed but a list of characters, see the example below.

should_be_string(*item*, *msg=None*)
Fails if the given `item` is not a string.

With Python 2, except with IronPython, this keyword passes regardless is the `item` a Unicode string or a byte string. Use *Should Be Unicode String* or *Should Be Byte String* if you want to restrict the string type. Notice that with Python 2, except with IronPython, `'string'` creates a byte string and `u'unicode'` must be used to create a Unicode string.

With Python 3 and IronPython, this keyword passes if the string is a Unicode string but fails if it is bytes. Notice that with both Python 3 and IronPython, `'string'` creates a Unicode string, and `b'bytes'` must be used to create a byte string.

The default error message can be overridden with the optional `msg` argument.

should_not_be_string(*item*, *msg=None*)
Fails if the given `item` is a string.

See *Should Be String* for more details about Unicode strings and byte strings.

The default error message can be overridden with the optional `msg` argument.

should_be_unicode_string(*item*, *msg=None*)
Fails if the given `item` is not a Unicode string.

Use *Should Be Byte String* if you want to verify the `item` is a byte string, or *Should Be String* if both Unicode and byte strings are fine. See *Should Be String* for more details about Unicode strings and byte strings.

The default error message can be overridden with the optional `msg` argument.

should_be_byte_string(*item*, *msg=None*)
Fails if the given `item` is not a byte string.

Use *Should Be Unicode String* if you want to verify the `item` is a Unicode string, or *Should Be String* if both Unicode and byte strings are fine. See *Should Be String* for more details about Unicode strings and byte strings.

The default error message can be overridden with the optional `msg` argument.

should_be_lowercase(*string*, *msg=None*)
Fails if the given `string` is not in lowercase.

For example, `'string'` and `'with specials!'` would pass, and `'String'`, `''` and `' '` would fail.

The default error message can be overridden with the optional `msg` argument.

See also *Should Be Uppercase* and *Should Be Titlecase*.

should_be_uppercase(*string*, *msg=None*)
Fails if the given `string` is not in uppercase.

For example, `'STRING'` and `'WITH SPECIALS!'` would pass, and `'String'`, `''` and `' '` would fail.

The default error message can be overridden with the optional `msg` argument.

See also *Should Be Titlecase* and *Should Be Lowercase*.

should_be_title_case(*string*, *msg=None*, *exclude=None*)
Fails if given `string` is not title.

`string` is a title cased string if there is at least one uppercase letter in each word.

For example, `'This Is Title'` and `'OK, Give Me My iPhone'` would pass. `'all words lower'` and `'Word In lower'` would fail.

This logic changed in Robot Framework 4.0 to be compatible with *Convert to Title Case*. See *Convert to Title Case* for title case algorithm and reasoning.

The default error message can be overridden with the optional `msg` argument.

Words can be explicitly excluded with the optional `exclude` argument.

Explicitly excluded words can be given as a list or as a string with words separated by a comma and an optional space. Excluded words are actually considered to be regular expression patterns, so it is possible to use something like "example[.!?]?" to match the word "example" on it own and also if followed by ".", "!" or "?". See *BuiltIn.Should Match Regexp* for more information about Python regular expression syntax in general and how to use it in Robot Framework test data in particular.

See also *Should Be Uppercase* and *Should Be Lowercase*.

robot.libraries.Telnet module

class robot.libraries.Telnet.**Telnet**(*timeout='3 seconds'*, *newline='CRLF'*, *prompt=None*, *prompt_is_regexp=False*, *encoding='UTF-8'*, *encoding_errors='ignore'*, *default_log_level='INFO'*, *window_size=None*, *environ_user=None*, *terminal_emulation=False*, *terminal_type=None*, *telnetlib_log_level='TRACE'*, *connection_timeout=None*)

Bases: `object`

A test library providing communication over Telnet connections.

`Telnet` is Robot Framework's standard library that makes it possible to connect to Telnet servers and execute commands on the opened connections.

== Table of contents ==

%TOC%

= Connections =

The first step of using `Telnet` is opening a connection with *Open Connection* keyword. Typically the next step is logging in with *Login* keyword, and in the end the opened connection can be closed with *Close Connection*.

It is possible to open multiple connections and switch the active one using *Switch Connection*. *Close All Connections* can be used to close all the connections, which is especially useful in suite teardowns to guarantee that all connections are always closed.

= Writing and reading =

After opening a connection and possibly logging in, commands can be executed or text written to the connection for other reasons using *Write* and *Write Bare* keywords. The main difference between these two is that the former adds a [#Configuration|configurable newline] after the text automatically.

After writing something to the connection, the resulting output can be read using *Read*, *Read Until*, *Read Until Regexp*, and *Read Until Prompt* keywords. Which one to use depends on the context, but the latest one is often the most convenient.

As a convenience when running a command, it is possible to use *Execute Command* that simply uses *Write* and *Read Until Prompt* internally. *Write Until Expected Output* is useful if you need to wait until writing something produces a desired output.

Written and read text is automatically encoded/decoded using a [#Configuration|configured encoding].

The ANSI escape codes, like cursor movement and color codes, are normally returned as part of the read operation. If an escape code occurs in middle of a search pattern it may also prevent finding the searched string. *Terminal emulation* can be used to process these escape codes as they would be if a real terminal would be in use.

= Configuration =

Many aspects related the connections can be easily configured either globally or per connection basis. Global configuration is done when [#Importing|library is imported], and these values can be overridden per connection

by *Open Connection* or with setting specific keywords *Set Timeout*, *Set Newline*, *Set Prompt*, *Set Encoding*, *Set Default Log Level* and *Set Telnetlib Log Level*.

Values of `environ_user`, `window_size`, `terminal_emulation`, and `terminal_type` can not be changed after opening the connection.

== Timeout ==

Timeout defines how long is the maximum time to wait when reading output. It is used internally by *Read Until*, *Read Until Regexp*, *Read Until Prompt*, and *Login* keywords. The default value is 3 seconds.

== Connection Timeout ==

Connection Timeout defines how long is the maximum time to wait when opening the telnet connection. It is used internally by *Open Connection*. The default value is the system global default timeout.

== Newline ==

Newline defines which line separator *Write* keyword should use. The default value is `CRLF` that is typically used by Telnet connections.

Newline can be given either in escaped format using `\n` and `\r` or with special `LF` and `CR` syntax.

== Prompt ==

Often the easiest way to read the output of a command is reading all the output until the next prompt with *Read Until Prompt*. It also makes it easier, and faster, to verify did *Login* succeed.

Prompt can be specified either as a normal string or a regular expression. The latter is especially useful if the prompt changes as a result of the executed commands. Prompt can be set to be a regular expression by giving `prompt_is_regexp` argument a true value (see *Boolean arguments*).

== Encoding ==

To ease handling text containing non-ASCII characters, all written text is encoded and read text decoded by default. The default encoding is UTF-8 that works also with ASCII. Encoding can be disabled by using a special encoding value `NONE`. This is mainly useful if you need to get the bytes received from the connection as-is.

Notice that when writing to the connection, only Unicode strings are encoded using the defined encoding. Byte strings are expected to be already encoded correctly. Notice also that normal text in test data is passed to the library as Unicode and you need to use variables to use bytes.

It is also possible to configure the error handler to use if encoding or decoding characters fails. Accepted values are the same that encode/decode functions in Python strings accept. In practice the following values are the most useful:

- `ignore`: ignore characters that cannot be encoded (default)
- `strict`: fail if characters cannot be encoded
- `replace`: replace characters that cannot be encoded with a replacement character

== Default log level ==

Default log level specifies the log level keywords use for *logging* unless they are given an explicit log level. The default value is `INFO`, and changing it, for example, to `DEBUG` can be a good idea if there is lot of unnecessary output that makes log files big.

== Terminal type ==

By default the Telnet library does not negotiate any specific terminal type with the server. If a specific terminal type, for example `vt100`, is desired, the terminal type can be configured in *importing* and with *Open Connection*.

== Window size ==

Window size for negotiation with the server can be configured when *importing* the library and with *Open Connection*.

== USER environment variable ==

Telnet protocol allows the `USER` environment variable to be sent when connecting to the server. On some servers it may happen that there is no login prompt, and on those cases this configuration option will allow still to define the desired username. The option `environ_user` can be used in *importing* and with *Open Connection*.

= Terminal emulation =

Telnet library supports terminal emulation with [http://pyte.readthedocs.io|Pyte]. Terminal emulation will process the output in a virtual screen. This means that ANSI escape codes, like cursor movements, and also control characters, like carriage returns and backspaces, have the same effect on the result as they would have on a normal terminal screen. For example the sequence `acdc\x1b[3Dbba` will result in output `abba`.

Terminal emulation is taken into use by giving `terminal_emulation` argument a true value (see *Boolean arguments*) either in the library initialization or with *Open Connection*.

As Pyte approximates vt-style terminal, you may also want to set the terminal type as `vt100`. We also recommend that you increase the window size, as the terminal emulation will break all lines that are longer than the window row length.

When terminal emulation is used, the *newline* and *encoding* can not be changed anymore after opening the connection.

As a prerequisite for using terminal emulation, you need to have Pyte installed. Due to backwards incompatible changes in Pyte, different Robot Framework versions support different Pyte versions:

- Pyte 0.6 and newer are supported by Robot Framework 3.0.3. Latest Pyte version can be installed (or upgraded) with `pip install --upgrade pyte`.
- Pyte 0.5.2 and older are supported by Robot Framework 3.0.2 and earlier. Pyte 0.5.2 can be installed with `pip install pyte==0.5.2`.

= Logging =

All keywords that read something log the output. These keywords take the log level to use as an optional argument, and if no log level is specified they use the [#Configuration|configured] default value.

The valid log levels to use are `TRACE`, `DEBUG`, `INFO` (default), and `WARN`. Levels below `INFO` are not shown in log files by default whereas warnings are shown more prominently.

The [http://docs.python.org/library/telnetlib.html|telnetlib module] used by this library has a custom logging system for logging content it sends and receives. By default these messages are written using `TRACE` level, but the level is configurable with the `telnetlib_log_level` option either in the library initialization, to the *Open Connection* or by using the *Set Telnetlib Log Level* keyword to the active connection. Special level `NONE` con be used to disable the logging altogether.

= Time string format =

Timeouts and other times used must be given as a time string using format like `15 seconds` or `1min 10s`. If the timeout is given as just a number, for example, `10` or `1.5`, it is considered to be seconds. The time string format is described in more detail in an appendix of [http://robotframework.org/robotframework/#user-guide|Robot Framework User Guide].

= Boolean arguments =

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-

insensitively. Other strings are considered true regardless their value, and other argument types are tested using the same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

True examples:

False examples:

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

Telnet library can be imported with optional configuration parameters.

Configuration parameters are used as default values when new connections are opened with *Open Connection* keyword. They can also be overridden after opening the connection using the *Set ...* *keywords*. See these keywords as well as *Configuration*, *Terminal emulation* and *Logging* sections above for more information about these parameters and their possible values.

See *Time string format* and *Boolean arguments* sections for information about using arguments accepting times and Boolean values, respectively.

ROBOT_LIBRARY_SCOPE = 'TEST_SUITE'

ROBOT_LIBRARY_VERSION = '4.0.3'

get_keyword_names()

open_connection(*host*, *alias=None*, *port=23*, *timeout=None*, *newline=None*, *prompt=None*, *prompt_is_regexp=False*, *encoding=None*, *encoding_errors=None*, *default_log_level=None*, *window_size=None*, *environ_user=None*, *terminal_emulation=None*, *terminal_type=None*, *telnetlib_log_level=None*, *connection_timeout=None*)

Opens a new Telnet connection to the given host and port.

The `timeout`, `newline`, `prompt`, `prompt_is_regexp`, `encoding`, `default_log_level`, `window_size`, `environ_user`, `terminal_emulation`, `terminal_type` and `telnetlib_log_level` arguments get default values when the library is [#Importing|imported]. Setting them here overrides those values for the opened connection. See *Configuration*, *Terminal emulation* and *Logging* sections for more information about these parameters and their possible values.

Possible already opened connections are cached and it is possible to switch back to them using *Switch Connection* keyword. It is possible to switch either using explicitly given `alias` or using index returned by this keyword. Indexing starts from 1 and is reset back to it by *Close All Connections* keyword.

switch_connection(*index_or_alias*)

Switches between active connections using an index or an alias.

Aliases can be given to *Open Connection* keyword which also always returns the connection index.

This keyword returns the index of previous active connection.

The example above expects that there were no other open connections when opening the first one, because it used index `1` when switching to the connection later. If you are not sure about that, you can store the index into a variable as shown below.

close_all_connections()

Closes all open connections and empties the connection cache.

If multiple connections are opened, this keyword should be used in a test or suite teardown to make sure that all connections are closed. It is not an error is some of the connections have already been closed by *Close Connection*.

After this keyword, new indexes returned by *Open Connection* keyword are reset to 1.

class robot.libraries.Telnet.**TelnetConnection**(*host=None*, *port=23*, *timeout=3.0*, *newline='CRLF'*, *prompt=None*, *prompt_is_regexp=False*, *encoding='UTF-8'*, *encoding_errors='ignore'*, *default_log_level='INFO'*, *window_size=None*, *environ_user=None*, *terminal_emulation=False*, *terminal_type=None*, *telnetlib_log_level='TRACE'*, *connection_timeout=None*)

Bases: telnetlib.Telnet

NEW_ENVIRON_IS = '\x00'

NEW_ENVIRON_VAR = '\x00'

NEW_ENVIRON_VALUE = '\x01'

INTERNAL_UPDATE_FREQUENCY = 0.03

set_timeout(*timeout*)

Sets the timeout used for waiting output in the current connection.

Read operations that expect some output to appear (*Read Until*, *Read Until Regexp*, *Read Until Prompt*, *Login*) use this timeout and fail if the expected output does not appear before this timeout expires.

The `timeout` must be given in *time string format*. The old timeout is returned and can be used to restore the timeout later.

See *Configuration* section for more information about global and connection specific configuration.

set_newline(*newline*)

Sets the newline used by *Write* keyword in the current connection.

The old newline is returned and can be used to restore the newline later. See *Set Timeout* for a similar example.

If terminal emulation is used, the newline can not be changed on an open connection.

See *Configuration* section for more information about global and connection specific configuration.

set_prompt(*prompt*, *prompt_is_regexp=False*)

Sets the prompt used by *Read Until Prompt* and *Login* in the current connection.

If `prompt_is_regexp` is given a true value (see *Boolean arguments*), the given `prompt` is considered to be a regular expression.

The old prompt is returned and can be used to restore the prompt later.

See the documentation of [http://docs.python.org/library/re.html|Python re module] for more information about the supported regular expression syntax. Notice that possible backslashes need to be escaped in Robot Framework test data.

See *Configuration* section for more information about global and connection specific configuration.

set_encoding(*encoding=None*, *errors=None*)

Sets the encoding to use for *writing and reading* in the current connection.

The given `encoding` specifies the encoding to use when written/read text is encoded/decoded, and `errors` specifies the error handler to use if encoding/decoding fails. Either of these can be omitted and in that case the old value is not affected. Use string `NONE` to disable encoding altogether.

See *Configuration* section for more information about encoding and error handlers, as well as global and connection specific configuration in general.

The old values are returned and can be used to restore the encoding and the error handler later. See *Set Prompt* for a similar example.

If terminal emulation is used, the encoding can not be changed on an open connection.

set_telnetlib_log_level (*level*)
Sets the log level used for *logging* in the underlying `telnetlib`.

Note that `telnetlib` can be very noisy thus using the level `NONE` can shutdown the messages generated by this library.

set_default_log_level (*level*)
Sets the default log level used for *logging* in the current connection.

The old default log level is returned and can be used to restore the log level later.

See *Configuration* section for more information about global and connection specific configuration.

close_connection (*loglevel=None*)
Closes the current Telnet connection.

Remaining output in the connection is read, logged, and returned. It is not an error to close an already closed connection.

Use *Close All Connections* if you want to make sure all opened connections are closed.

See *Logging* section for more information about log levels.

login (*username*, *password*, *login_prompt='login: '*, *password_prompt='Password: '*, *login_timeout='1 second'*, *login_incorrect='Login incorrect'*)
Logs in to the Telnet server with the given user information.

This keyword reads from the connection until the `login_prompt` is encountered and then types the given `username`. Then it reads until the `password_prompt` and types the given `password`. In both cases a newline is appended automatically and the connection specific timeout used when waiting for outputs.

How logging status is verified depends on whether a prompt is set for this connection or not:

1) If the prompt is set, this keyword reads the output until the prompt is found using the normal timeout. If no prompt is found, login is considered failed and also this keyword fails. Note that in this case both `login_timeout` and `login_incorrect` arguments are ignored.

2) If the prompt is not set, this keywords sleeps until `login_timeout` and then reads all the output available on the connection. If the output contains `login_incorrect` text, login is considered failed and also this keyword fails.

See *Configuration* section for more information about setting newline, timeout, and prompt.

write (*text*, *loglevel=None*)
Writes the given text plus a newline into the connection.

The newline character sequence to use can be [#Configuration|configured] both globally and per connection basis. The default value is `CRLF`.

This keyword consumes the written text, until the added newline, from the output and logs and returns it. The given text itself must not contain newlines. Use *Write Bare* instead if either of these features causes a problem.

Note: This keyword does not return the possible output of the executed command. To get the output, one of the *Read ... keywords* must be used. See *Writing and reading* section for more details.

See *Logging* section for more information about log levels.

write_bare(*text*)
Writes the given text, and nothing else, into the connection.

This keyword does not append a newline nor consume the written text. Use *Write* if these features are needed.

write_until_expected_output(*text*, *expected*, *timeout*, *retry_interval*, *loglevel=None*)
Writes the given `text` repeatedly, until `expected` appears in the output.

`text` is written without appending a newline and it is consumed from the output before trying to find `expected`. If `expected` does not appear in the output within `timeout`, this keyword fails.

`retry_interval` defines the time to wait `expected` to appear before writing the `text` again. Consuming the written `text` is subject to the normal [#Configuration|configured timeout].

Both `timeout` and `retry_interval` must be given in *time string format*. See *Logging* section for more information about log levels.

The above example writes command `ps -ef | grep myprocess\r\n` until `myprocess` appears in the output. The command is written every 0.5 seconds and the keyword fails if `myprocess` does not appear in the output in 5 seconds.

write_control_character(*character*)
Writes the given control character into the connection.

The control character is prepended with an IAC (interpret as command) character.

The following control character names are supported: BRK, IP, AO, AYT, EC, EL, NOP. Additionally, you can use arbitrary numbers to send any control character.

read(*loglevel=None*)
Reads everything that is currently available in the output.

Read output is both returned and logged. See *Logging* section for more information about log levels.

read_until(*expected*, *loglevel=None*)
Reads output until `expected` text is encountered.

Text up to and including the match is returned and logged. If no match is found, this keyword fails. How much to wait for the output depends on the [#Configuration|configured timeout].

See *Logging* section for more information about log levels. Use *Read Until Regexp* if more complex matching is needed.

read_until_regexp(**expected*)
Reads output until any of the `expected` regular expressions match.

This keyword accepts any number of regular expressions patterns or compiled Python regular expression objects as arguments. Text up to and including the first match to any of the regular expressions is returned and logged. If no match is found, this keyword fails. How much to wait for the output depends on the [#Configuration|configured timeout].

If the last given argument is a [#Logging|valid log level], it is used as `loglevel` similarly as with *Read Until* keyword.

See the documentation of [http://docs.python.org/library/re.html|Python re module] for more information about the supported regular expression syntax. Notice that possible backslashes need to be escaped in Robot Framework test data.

read_until_prompt(*loglevel=None*, *strip_prompt=False*)
Reads output until the prompt is encountered.

This keyword requires the prompt to be [#Configuration|configured] either in *importing* or with *Open Connection* or *Set Prompt* keyword.

By default, text up to and including the prompt is returned and logged. If no prompt is found, this keyword fails. How much to wait for the output depends on the [#Configuration|configured timeout].

If you want to exclude the prompt from the returned output, set `strip_prompt` to a true value (see *Boolean arguments*). If your prompt is a regular expression, make sure that the expression spans the whole prompt, because only the part of the output that matches the regular expression is stripped away.

See *Logging* section for more information about log levels.

execute_command(*command*, *loglevel=None*, *strip_prompt=False*)
Executes the given `command` and reads, logs, and returns everything until the prompt.

This keyword requires the prompt to be [#Configuration|configured] either in *importing* or with *Open Connection* or *Set Prompt* keyword.

This is a convenience keyword that uses *Write* and *Read Until Prompt* internally. Following two examples are thus functionally identical:

See *Logging* section for more information about log levels and *Read Until Prompt* for more information about the `strip_prompt` parameter.

msg(*msg*, **args*)

close()
Close the connection.

expect(*list*, *timeout=None*)
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (re.RegexObject instances) or uncompiled (strings). The optional second argument is a timeout, in seconds; default is no timeout.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If EOF is read and no text was read, raise EOFError. Otherwise, when nothing matches, return (-1, None, text) where text is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (e.g. '.*') or if more than one expression can match the same input, the results are undeterministic, and may depend on the I/O timing.

fileno()
Return the fileno() of the socket object used internally.

fill_rawq()
Fill raw queue from exactly one recv() system call.

Block if no data is immediately available. Set self.eof when connection is closed.

get_socket()
Return the socket object used internally.

interact()
Interaction function, emulates a very dumb telnet client.

listener()
Helper for mt_interact() – this executes in the other thread.

mt_interact()
Multithreaded version of interact().

open(*host*, *port=0*, *timeout=<object object>*)
Connect to a host.

The optional second argument is the port number, which defaults to the standard telnet port (23).

Don't try to reopen an already connected instance.

process_rawq()
Transfer from raw queue to cooked queue.

Set self.eof when connection is closed. Don't block unless in the midst of an IAC sequence.

rawq_getchar()
Get next char from raw queue.

Block if no data is immediately available. Raise EOFError when connection is closed.

read_all()
Read all data until EOF; block until connection closed.

read_eager()
Read readily available data.

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_lazy()
Process and return data that's already in the queues (lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_sb_data()
Return any data available in the SB ... SE queue.

Return '' if no SB ... SE available. Should only be called after seeing a SB or SE command. When a new SB command is found, old unread SB data will be discarded. Don't block.

read_some()
Read at least one byte of cooked data unless EOF is hit.

Return '' if EOF is hit. Block if no data is immediately available.

read_very_eager()
Read everything that's possible without blocking in I/O (eager).

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_very_lazy()
Return any data available in the cooked queue (very lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Don't block.

set_debuglevel(*debuglevel*)
Set the debug level.

The higher it is, the more debug output you get (on sys.stdout).

set_option_negotiation_callback(*callback*)
Provide a callback function called after each receipt of a telnet option.

sock_avail()
Test whether data is available on the socket.

class robot.libraries.Telnet.**TerminalEmulator**(*window_size=None*, *newline='rn'*)
Bases: object

current_output

feed(*text*)

read()

read_until(*expected*)

read_until_regexp(*regexp_list*)

exception robot.libraries.Telnet.**NoMatchError**(*expected*, *timeout*, *output=None*)
Bases: exceptions.AssertionError

ROBOT_SUPPRESS_NAME = True

args

message

robot.libraries.XML module

class robot.libraries.XML.**XML**(*use_lxml=False*)
Bases: object

Robot Framework test library for verifying and modifying XML documents.

As the name implies, _XML_ is a test library for verifying contents of XML files. In practice it is a pretty thin wrapper on top of Python's [http://docs.python.org/library/xml.etree.elementtree.html|ElementTree XML API].

The library has the following main usages:

- Parsing an XML file, or a string containing XML, into an XML element structure and finding certain elements from it for for further analysis (e.g. *Parse XML* and *Get Element* keywords).
- Getting text or attributes of elements (e.g. *Get Element Text* and *Get Element Attribute*).
- Directly verifying text, attributes, or whole elements (e.g *Element Text Should Be* and *Elements Should Be Equal*).
- Modifying XML and saving it (e.g. *Set Element Text*, *Add Element* and *Save XML*).

== Table of contents ==

%TOC%

= Parsing XML =

XML can be parsed into an element structure using *Parse XML* keyword. The XML to be parsed can be specified using a path to an XML file or as a string or bytes that contain XML directly. The keyword returns the root element of the structure, which then contains other elements as its children and their children. Possible comments and processing instructions in the source XML are removed.

XML is not validated during parsing even if has a schema defined. How possible doctype elements are handled otherwise depends on the used XML module and on the platform. The standard ElementTree strips doctypes altogether but when *using lxml* they are preserved when XML is saved.

The element structure returned by *Parse XML*, as well as elements returned by keywords such as *Get Element*, can be used as the `source` argument with other keywords. In addition to an already parsed XML structure, other keywords also accept paths to XML files and strings containing XML similarly as *Parse XML*. Notice that keywords that modify XML do not write those changes back to disk even if the source would be given as a path to a file. Changes must always saved explicitly using *Save XML* keyword.

When the source is given as a path to a file, the forward slash character (`/`) can be used as the path separator regardless the operating system. On Windows also the backslash works, but it the test data it needs to be escaped by doubling it (`\\`). Using the built-in variable `${/}` naturally works too.

Note: Support for XML as bytes is new in Robot Framework 3.2.

= Using lxml =

By default this library uses Python's standard [http://docs.python.org/library/xml.etree.elementtree.html|ElementTree] module for parsing XML, but it can be configured to use [http://lxml.de|lxml] module instead when *importing* the library. The resulting element structure has same API regardless which module is used for parsing.

The main benefits of using lxml is that it supports richer xpath syntax than the standard ElementTree and enables using *Evaluate Xpath* keyword. It also preserves the doctype and possible namespace prefixes saving XML.

= Example =

The following simple example demonstrates parsing XML and verifying its contents both using keywords in this library and in _BuiltIn_ and _Collections_ libraries. How to use xpath expressions to find elements and what attributes the returned elements contain are discussed, with more examples, in *Finding elements with xpath* and *Element attributes* sections.

In this example, as well as in many other examples in this documentation, `${XML}` refers to the following example XML document. In practice `${XML}` could either be a path to an XML file or it could contain the XML itself.

Notice that in the example three last lines are equivalent. Which one to use in practice depends on which other elements you need to get or verify. If you only need to do one verification, using the last line alone would suffice. If more verifications are needed, parsing the XML with *Parse XML* only once would be more efficient.

= Finding elements with xpath =

ElementTree, and thus also this library, supports finding elements using xpath expressions. ElementTree does not, however, support the full xpath standard. The supported xpath syntax is explained below and [https://docs.python.org/library/xml.etree.elementtree.html#xpath-support| ElementTree documentation] provides more details. In the examples `${XML}` refers to the same XML structure as in the earlier example.

If lxml support is enabled when *importing* the library, the whole [http://www.w3.org/TR/xpath/|xpath 1.0 standard] is supported. That includes everything listed below but also lot of other useful constructs.

== Tag names ==

When just a single tag name is used, xpath matches all direct child elements that have that tag name.

== Paths ==

Paths are created by combining tag names with a forward slash (`/`). For example, `parent/child` matches all `child` elements under `parent` element. Notice that if there are multiple `parent` elements that all have `child` elements, `parent/child` xpath will match all these `child` elements.

== Wildcards ==

An asterisk (`*`) can be used in paths instead of a tag name to denote any element.

== Current element ==

The current element is denoted with a dot (`.`). Normally the current element is implicit and does not need to be included in the xpath.

== Parent element ==

The parent element of another element is denoted with two dots (`..`). Notice that it is not possible to refer to the parent of the current element.

== Search all sub elements ==

Two forward slashes (`//`) mean that all sub elements, not only the direct children, are searched. If the search is started from the current element, an explicit dot is required.

== Predicates ==

Predicates allow selecting elements using also other criteria than tag names, for example, attributes or position. They are specified after the normal tag name or path using syntax `path[predicate]`. The path can have wildcards and other special syntax explained earlier. What predicates the standard ElementTree supports is explained in the table below.

Predicates can also be stacked like `path[predicate1][predicate2]`. A limitation is that possible position predicate must always be first.

= Element attributes =

All keywords returning elements, such as *Parse XML*, and *Get Element*, return ElementTree's [http://docs.python.org/library/xml.etree.elementtree.html#element-objects|Element objects]. These elements can be used as inputs for other keywords, but they also contain several useful attributes that can be accessed directly using the extended variable syntax.

The attributes that are both useful and convenient to use in the test data are explained below. Also other attributes, including methods, can be accessed, but that is typically better to do in custom libraries than directly in the test data.

The examples use the same `${XML}` structure as the earlier examples.

== tag ==

The tag of the element.

== text ==

The text that the element contains or Python `None` if the element has no text. Notice that the text _does **not**_ contain texts of possible child elements nor text after or between children. Notice also that in XML whitespace is significant, so the text contains also possible indentation and newlines. To get also text of the possible children, optionally whitespace normalized, use *Get Element Text* keyword.

== tail ==

The text after the element before the next opening or closing tag. Python `None` if the element has no tail. Similarly as with `text`, also `tail` contains possible indentation and newlines.

== attrib ==

A Python dictionary containing attributes of the element.

= Handling XML namespaces =

ElementTree and lxml handle possible namespaces in XML documents by adding the namespace URI to tag names in so called Clark Notation. That is inconvenient especially with xpaths, and by default this library strips those namespaces away and moves them to `xmlns` attribute instead. That can be avoided by passing `keep_clark_notation` argument to *Parse XML* keyword. Alternatively *Parse XML* supports stripping namespace information altogether by using `strip_namespaces` argument. The pros and cons of different approaches are discussed in more detail below.

== How ElementTree handles namespaces ==

If an XML document has namespaces, ElementTree adds namespace information to tag names in [http://www.jclark.com/xml/xmlns.htm|Clark Notation] (e.g. `{http://ns.uri}tag`) and removes original `xmlns` attributes. This is done both with default namespaces and with namespaces with a prefix. How it works in practice is illustrated by the following example, where `${NS}` variable contains this XML document:

As you can see, including the namespace URI in tag names makes xpaths really long and complex.

If you save the XML, ElementTree moves namespace information back to `xmlns` attributes. Unfortunately it does not restore the original prefixes:

The resulting output is semantically same as the original, but mangling prefixes like this may still not be desirable. Notice also that the actual output depends slightly on ElementTree version.

== Default namespace handling ==

Because the way ElementTree handles namespaces makes xpaths so complicated, this library, by default, strips namespaces from tag names and moves that information back to `xmlns` attributes. How this works in practice is shown by the example below, where `${NS}` variable contains the same XML document as in the previous example.

Now that tags do not contain namespace information, xpaths are simple again.

A minor limitation of this approach is that namespace prefixes are lost. As a result the saved output is not exactly same as the original one in this case either:

Also this output is semantically same as the original. If the original XML had only default namespaces, the output would also look identical.

== Namespaces when using lxml ==

This library handles namespaces same way both when *using lxml* and when not using it. There are, however, differences how lxml internally handles namespaces compared to the standard ElementTree. The main difference is that lxml stores information about namespace prefixes and they are thus preserved if XML is saved. Another visible difference is that lxml includes namespace information in child elements got with *Get Element* if the parent element has namespaces.

== Stripping namespaces altogether ==

Because namespaces often add unnecessary complexity, *Parse XML* supports stripping them altogether by using `strip_namespaces=True`. When this option is enabled, namespaces are not shown anywhere nor are they included if XML is saved.

== Attribute namespaces ==

Attributes in XML documents are, by default, in the same namespaces as the element they belong to. It is possible to use different namespaces by using prefixes, but this is pretty rare.

If an attribute has a namespace prefix, ElementTree will replace it with Clark Notation the same way it handles elements. Because stripping namespaces from attributes could cause attribute conflicts, this library does not handle attribute namespaces at all. Thus the following example works the same way regardless how namespaces are handled.

= Boolean arguments =

Some keywords accept arguments that are handled as Boolean values true or false. If such an argument is given as a string, it is considered false if it is an empty string or equal to `FALSE`, `NONE`, `NO`, `OFF` or `0`, case-insensitively. Other strings are considered true regardless their value, and other argument types are tested using the same [http://docs.python.org/library/stdtypes.html#truth|rules as in Python].

True examples:

False examples:

Considering `OFF` and `0` false is new in Robot Framework 3.1.

== Pattern matching ==

Some keywords, for example *Elements Should Match*, support so called [http://en.wikipedia.org/wiki/Glob_(programming)|glob patterns] where:

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

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

Import library with optionally lxml mode enabled.

By default this library uses Python's standard [http://docs.python.org/library/xml.etree.elementtree.html|ElementTree] module for parsing XML. If use_lxml argument is given a true value (see *Boolean arguments*), the library will use [http://lxml.de|lxml] module instead. See *Using lxml* section for benefits provided by lxml.

Using lxml requires that the lxml module is installed on the system. If lxml mode is enabled but the module is not installed, this library will emit a warning and revert back to using the standard ElementTree.

ROBOT_LIBRARY_SCOPE = 'GLOBAL'

ROBOT_LIBRARY_VERSION = '4.0.3'

parse_xml(*source*, *keep_clark_notation=False*, *strip_namespaces=False*)

Parses the given XML file or string into an element structure.

The source can either be a path to an XML file or a string containing XML. In both cases the XML is parsed into ElementTree [http://docs.python.org/library/xml.etree.elementtree.html#element-objects|element structure] and the root element is returned. Possible comments and processing instructions in the source XML are removed.

As discussed in *Handling XML namespaces* section, this keyword, by default, removes namespace information ElementTree has added to tag names and moves it into xmlns attributes. This typically eases handling XML documents with namespaces considerably. If you do not want that to happen, or want to avoid the small overhead of going through the element structure when your XML does not have namespaces, you can disable this feature by giving keep_clark_notation argument a true value (see *Boolean arguments*).

If you want to strip namespace information altogether so that it is not included even if XML is saved, you can give a true value to strip_namespaces argument.

Use *Get Element* keyword if you want to get a certain element and not the whole structure. See *Parsing XML* section for more details and examples.

get_element(*source*, *xpath='.'*)

Returns an element in the source matching the xpath.

The source can be a path to an XML file, a string containing XML, or an already parsed XML element. The xpath specifies which element to find. See the *introduction* for more details about both the possible sources and the supported xpath syntax.

The keyword fails if more, or less, than one element matches the xpath. Use *Get Elements* if you want all matching elements to be returned.

Parse XML is recommended for parsing XML when the whole structure is needed. It must be used if there is a need to configure how XML namespaces are handled.

Many other keywords use this keyword internally, and keywords modifying XML are typically documented to both to modify the given source and to return it. Modifying the source does not apply if the source is given as a string. The XML structure parsed based on the string and then modified is nevertheless returned.

get_elements(*source*, *xpath*)

Returns a list of elements in the source matching the xpath.

The source can be a path to an XML file, a string containing XML, or an already parsed XML element. The xpath specifies which element to find. See the *introduction* for more details.

Elements matching the `xpath` are returned as a list. If no elements match, an empty list is returned. Use *Get Element* if you want to get exactly one match.

get_child_elements (*source*, *xpath='.'*)
Returns the child elements of the specified element as a list.

The element whose children to return is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

All the direct child elements of the specified element are returned. If the element has no children, an empty list is returned.

get_element_count (*source*, *xpath='.'*)
Returns and logs how many elements the given `xpath` matches.

Arguments `source` and `xpath` have exactly the same semantics as with *Get Elements* keyword that this keyword uses internally.

See also *Element Should Exist* and *Element Should Not Exist*.

element_should_exist (*source*, *xpath='.'*, *message=None*)
Verifies that one or more element match the given `xpath`.

Arguments `source` and `xpath` have exactly the same semantics as with *Get Elements* keyword. Keyword passes if the `xpath` matches one or more elements in the `source`. The default error message can be overridden with the `message` argument.

See also *Element Should Not Exist* as well as *Get Element Count* that this keyword uses internally.

element_should_not_exist (*source*, *xpath='.'*, *message=None*)
Verifies that no element match the given `xpath`.

Arguments `source` and `xpath` have exactly the same semantics as with *Get Elements* keyword. Keyword fails if the `xpath` matches any element in the `source`. The default error message can be overridden with the `message` argument.

See also *Element Should Exist* as well as *Get Element Count* that this keyword uses internally.

get_element_text (*source*, *xpath='.'*, *normalize_whitespace=False*)
Returns all text of the element, possibly whitespace normalized.

The element whose text to return is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

This keyword returns all the text of the specified element, including all the text its children and grandchildren contain. If the element has no text, an empty string is returned. The returned text is thus not always the same as the *text* attribute of the element.

By default all whitespace, including newlines and indentation, inside the element is returned as-is. If `normalize_whitespace` is given a true value (see *Boolean arguments*), then leading and trailing whitespace is stripped, newlines and tabs converted to spaces, and multiple spaces collapsed into one. This is especially useful when dealing with HTML data.

See also *Get Elements Texts*, *Element Text Should Be* and *Element Text Should Match*.

get_elements_texts (*source*, *xpath*, *normalize_whitespace=False*)
Returns text of all elements matching `xpath` as a list.

The elements whose text to return is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Elements* keyword.

The text of the matched elements is returned using the same logic as with *Get Element Text*. This includes optional whitespace normalization using the `normalize_whitespace` option.

element_text_should_be(*source*, *expected*, *xpath='.'*, *normalize_whitespace=False*, *message=None*)
Verifies that the text of the specified element is `expected`.

The element whose text is verified is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

The text to verify is got from the specified element using the same logic as with *Get Element Text*. This includes optional whitespace normalization using the `normalize_whitespace` option.

The keyword passes if the text of the element is equal to the `expected` value, and otherwise it fails. The default error message can be overridden with the `message` argument. Use *Element Text Should Match* to verify the text against a pattern instead of an exact value.

element_text_should_match(*source*, *pattern*, *xpath='.'*, *normalize_whitespace=False*, *message=None*)
Verifies that the text of the specified element matches `expected`.

This keyword works exactly like *Element Text Should Be* except that the expected value can be given as a pattern that the text of the element must match.

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

get_element_attribute(*source*, *name*, *xpath='.'*, *default=None*)
Returns the named attribute of the specified element.

The element whose attribute to return is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

The value of the attribute `name` of the specified element is returned. If the element does not have such element, the `default` value is returned instead.

See also *Get Element Attributes*, *Element Attribute Should Be*, *Element Attribute Should Match* and *Element Should Not Have Attribute*.

get_element_attributes(*source*, *xpath='.'*)
Returns all attributes of the specified element.

The element whose attributes to return is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

Attributes are returned as a Python dictionary. It is a copy of the original attributes so modifying it has no effect on the XML structure.

Use *Get Element Attribute* to get the value of a single attribute.

element_attribute_should_be(*source*, *name*, *expected*, *xpath='.'*, *message=None*)
Verifies that the specified attribute is `expected`.

The element whose attribute is verified is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

The keyword passes if the attribute `name` of the element is equal to the `expected` value, and otherwise it fails. The default error message can be overridden with the `message` argument.

To test that the element does not have a certain attribute, Python `None` (i.e. variable `${NONE}`) can be used as the expected value. A cleaner alternative is using *Element Should Not Have Attribute*.

See also *Element Attribute Should Match* and *Get Element Attribute*.

element_attribute_should_match(*source*, *name*, *pattern*, *xpath='.'*, *message=None*)
Verifies that the specified attribute matches `expected`.

This keyword works exactly like *Element Attribute Should Be* except that the expected value can be given as a pattern that the attribute of the element must match.

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

element_should_not_have_attribute(*source*, *name*, *xpath='.'*, *message=None*)
Verifies that the specified element does not have attribute `name`.

The element whose attribute is verified is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

The keyword fails if the specified element has attribute `name`. The default error message can be overridden with the `message` argument.

See also *Get Element Attribute*, *Get Element Attributes*, *Element Text Should Be* and *Element Text Should Match*.

elements_should_be_equal(*source*, *expected*, *exclude_children=False*, *normalize_whitespace=False*)
Verifies that the given `source` element is equal to `expected`.

Both `source` and `expected` can be given as a path to an XML file, as a string containing XML, or as an already parsed XML element structure. See *introduction* for more information about parsing XML in general.

The keyword passes if the `source` element and `expected` element are equal. This includes testing the tag names, texts, and attributes of the elements. By default also child elements are verified the same way, but this can be disabled by setting `exclude_children` to a true value (see *Boolean arguments*).

All texts inside the given elements are verified, but possible text outside them is not. By default texts must match exactly, but setting `normalize_whitespace` to a true value makes text verification independent on newlines, tabs, and the amount of spaces. For more details about handling text see *Get Element Text* keyword and discussion about elements' *text* and *tail* attributes in the *introduction*.

The last example may look a bit strange because the `<p>` element only has text `Text with`. The reason is that rest of the text inside `<p>` actually belongs to the child elements. This includes the `.` at the end that is the *tail* text of the `<i>` element.

See also *Elements Should Match*.

elements_should_match(*source*, *expected*, *exclude_children=False*, *normalize_whitespace=False*)
Verifies that the given `source` element matches `expected`.

This keyword works exactly like *Elements Should Be Equal* except that texts and attribute values in the expected value can be given as patterns.

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

See *Elements Should Be Equal* for more examples.

set_element_tag(*source*, *tag*, *xpath='.'*)
Sets the tag of the specified element.

The element whose tag to set is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

Can only set the tag of a single element. Use *Set Elements Tag* to set the tag of multiple elements in one call.

set_elements_tag(*source*, *tag*, *xpath='.'*)
Sets the tag of the specified elements.

Like *Set Element Tag* but sets the tag of all elements matching the given `xpath`.

set_element_text(*source*, *text=None*, *tail=None*, *xpath='.'*)
Sets text and/or tail text of the specified element.

The element whose text to set is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

Element's text and tail text are changed only if new `text` and/or `tail` values are given. See *Element attributes* section for more information about *text* and *tail* in general.

Can only set the text/tail of a single element. Use *Set Elements Text* to set the text/tail of multiple elements in one call.

set_elements_text(*source*, *text=None*, *tail=None*, *xpath='.'*)
Sets text and/or tail text of the specified elements.

Like *Set Element Text* but sets the text or tail of all elements matching the given `xpath`.

set_element_attribute(*source*, *name*, *value*, *xpath='.'*)
Sets attribute `name` of the specified element to `value`.

The element whose attribute to set is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

It is possible to both set new attributes and to overwrite existing. Use *Remove Element Attribute* or *Remove Element Attributes* for removing them.

Can only set an attribute of a single element. Use *Set Elements Attribute* to set an attribute of multiple elements in one call.

set_elements_attribute(*source*, *name*, *value*, *xpath='.'*)
Sets attribute `name` of the specified elements to `value`.

Like *Set Element Attribute* but sets the attribute of all elements matching the given `xpath`.

remove_element_attribute(*source*, *name*, *xpath='.'*)
Removes attribute `name` from the specified element.

The element whose attribute to remove is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

It is not a failure to remove a non-existing attribute. Use *Remove Element Attributes* to remove all attributes and *Set Element Attribute* to set them.

Can only remove an attribute from a single element. Use *Remove Elements Attribute* to remove an attribute of multiple elements in one call.

remove_elements_attribute(*source*, *name*, *xpath='.'*)
Removes attribute `name` from the specified elements.

Like *Remove Element Attribute* but removes the attribute of all elements matching the given `xpath`.

remove_element_attributes(*source*, *xpath='.'*)
Removes all attributes from the specified element.

The element whose attributes to remove is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

Use *Remove Element Attribute* to remove a single attribute and *Set Element Attribute* to set them.

Can only remove attributes from a single element. Use *Remove Elements Attributes* to remove all attributes of multiple elements in one call.

remove_elements_attributes(*source*, *xpath='.'*)
Removes all attributes from the specified elements.

Like *Remove Element Attributes* but removes all attributes of all elements matching the given `xpath`.

add_element(*source*, *element*, *index=None*, *xpath='.'*)
Adds a child element to the specified element.

The element to whom to add the new element is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

The `element` to add can be specified as a path to an XML file or as a string containing XML, or it can be an already parsed XML element. The element is copied before adding so modifying either the original or the added element has no effect on the other . The element is added as the last child by default, but a custom index can be used to alter the position. Indices start from zero (0 = first position, 1 = second position, etc.), and negative numbers refer to positions at the end (-1 = second last position, -2 = third last, etc.).

Use *Remove Element* or *Remove Elements* to remove elements.

remove_element(*source*, *xpath=''*, *remove_tail=False*)
Removes the element matching `xpath` from the `source` structure.

The element to remove from the `source` is specified with `xpath` using the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

The keyword fails if `xpath` does not match exactly one element. Use *Remove Elements* to remove all matched elements.

Element's tail text is not removed by default, but that can be changed by giving `remove_tail` a true value (see *Boolean arguments*). See *Element attributes* section for more information about *tail* in general.

remove_elements(*source*, *xpath=''*, *remove_tail=False*)
Removes all elements matching `xpath` from the `source` structure.

The elements to remove from the `source` are specified with `xpath` using the same semantics as with *Get Elements* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

It is not a failure if `xpath` matches no elements. Use *Remove Element* to remove exactly one element.

Element's tail text is not removed by default, but that can be changed by using `remove_tail` argument similarly as with *Remove Element*.

clear_element(*source*, *xpath='.'*, *clear_tail=False*)
Clears the contents of the specified element.

The element to clear is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword. The resulting XML structure is returned, and if the `source` is an already parsed XML structure, it is also modified in place.

Clearing the element means removing its text, attributes, and children. Element's tail text is not removed by default, but that can be changed by giving `clear_tail` a true value (see *Boolean arguments*). See *Element attributes* section for more information about tail in general.

Use *Remove Element* to remove the whole element.

copy_element (*source*, *xpath='.'*)
Returns a copy of the specified element.

The element to copy is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

If the copy or the original element is modified afterwards, the changes have no effect on the other.

element_to_string (*source*, *xpath='.'*, *encoding=None*)
Returns the string representation of the specified element.

The element to convert to a string is specified using `source` and `xpath`. They have exactly the same semantics as with *Get Element* keyword.

By default the string is returned as Unicode. If `encoding` argument is given any value, the string is returned as bytes in the specified encoding. The resulting string never contains the XML declaration.

See also *Log Element* and *Save XML*.

log_element (*source*, *level='INFO'*, *xpath='.'*)
Logs the string representation of the specified element.

The element specified with `source` and `xpath` is first converted into a string using *Element To String* keyword internally. The resulting string is then logged using the given `level`.

The logged string is also returned.

save_xml (*source*, *path*, *encoding='UTF-8'*)
Saves the given element to the specified file.

The element to save is specified with `source` using the same semantics as with *Get Element* keyword.

The file where the element is saved is denoted with `path` and the encoding to use with `encoding`. The resulting file always contains the XML declaration.

The resulting XML file may not be exactly the same as the original: - Comments and processing instructions are always stripped. - Possible doctype and namespace prefixes are only preserved when

using lxml.

- Other small differences are possible depending on the ElementTree or lxml version.

Use *Element To String* if you just need a string representation of the element.

evaluate_xpath (*source*, *expression*, *context='.'*)
Evaluates the given xpath expression and returns results.

The element in which context the expression is executed is specified using `source` and `context` arguments. They have exactly the same semantics as `source` and `xpath` arguments have with *Get Element* keyword.

The xpath expression to evaluate is given as `expression` argument. The result of the evaluation is returned as-is.

This keyword works only if lxml mode is taken into use when *importing* the library.

class robot.libraries.XML.**NameSpaceStripper** (*etree*, *lxml_etree=False*)
Bases: `object`

strip(*elem*, *preserve=True*, *current_ns=None*, *top=True*)

unstrip(*elem*, *current_ns=None*, *copied=False*)

class robot.libraries.XML.**ElementFinder**(*etree*, *modern=True*, *lxml=False*)
Bases: object

find_all(*elem*, *xpath*)

class robot.libraries.XML.**ElementComparator**(*comparator*, *normalizer=None*, *exclude_children=False*)
Bases: object

compare(*actual*, *expected*, *location=None*)

class robot.libraries.XML.**Location**(*path*, *is_root=True*)
Bases: object

child(*tag*)

robot.libraries.dialogs_ipy module

robot.libraries.dialogs_jy module

robot.libraries.dialogs_py module

class robot.libraries.dialogs_py.**MessageDialog**(*message*, *value=None*, ***extra*)
Bases: robot.libraries.dialogs_py._TkDialog

after(*ms*, *func=None*, **args*)
Call function once after given time.

MS specifies the time in milliseconds. FUNC gives the function which shall be called. Additional parameters are given as parameters to the function call. Return identifier to cancel scheduling with after_cancel.

after_cancel(*id*)
Cancel scheduling of function identified with ID.

Identifier returned by after or after_idle must be given as first parameter.

after_idle(*func*, **args*)
Call FUNC once if the Tcl main loop has no event to process.

Return an identifier to cancel the scheduling with after_cancel.

aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

bbox (*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

bell (*displayof=0*)
Ring a display's bell.

bind (*sequence=None*, *func=None*, *add=None*)
Bind to this widget at event SEQUENCE a call to function FUNC.

SEQUENCE is a string of concatenated event patterns. An event pattern is of the form <MODIFIER-MODIFIER-TYPE-DETAIL> where MODIFIER is one of Control, Mod2, M2, Shift, Mod3, M3, Lock, Mod4, M4, Button1, B1, Mod5, M5 Button2, B2, Meta, M, Button3, B3, Alt, Button4, B4, Double, Button5, B5 Triple, Mod1, M1. TYPE is one of Activate, Enter, Map, ButtonPress, Button, Expose, Motion, ButtonRelease FocusIn, MouseWheel, Circulate, FocusOut, Property, Colormap, Gravity Reparent, Configure, KeyPress, Key, Unmap, Deactivate, KeyRelease Visibility, Destroy, Leave and DETAIL is the button number for ButtonPress, ButtonRelease and DETAIL is the Keysym for KeyPress and KeyRelease. Examples are <Control-Button-1> for pressing Control and mouse button 1 or <Alt-A> for pressing A and the Alt key (KeyPress can be omitted). An event pattern can also be a virtual event of the form <<AString>> where AString can be arbitrary. This event can be generated by event_generate. If events are concatenated they must appear shortly after each other.

FUNC will be called if the event sequence occurs with an instance of Event as argument. If the return value of FUNC is "break" no further bound function is invoked.

An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function.

Bind will return an identifier to allow deletion of the bound function with unbind without memory leak.

If FUNC or SEQUENCE is omitted the bound function or list of bound events are returned.

bind_all (*sequence=None*, *func=None*, *add=None*)
Bind to all widgets at an event SEQUENCE a call to function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bind_class (*className*, *sequence=None*, *func=None*, *add=None*)
Bind to widgets with bindtag CLASSNAME at event SEQUENCE a call of function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bindtags (*tagList=None*)
Set or get the list of bindtags for this widget.

With no argument return the list of all bindtags associated with this widget. With a list of strings as argument the bindtags are set to this list. The bindtags determine in which order events are processed (see bind).

cget (*key*)
Return the resource value for a KEY given as string.

client (*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

clipboard_append(*string*, ***kw*)
Append STRING to the Tk clipboard.

A widget specified at the optional displayof keyword argument specifies the target display. The clipboard can be retrieved with selection_get.

clipboard_clear(***kw*)
Clear the data in the Tk clipboard.

A widget specified for the optional displayof keyword argument specifies the target display.

clipboard_get(***kw*)
Retrieve data from the clipboard on window's display.

The window keyword defaults to the root window of the Tkinter application.

The type keyword specifies the form in which the data is to be returned and should be an atom name such as STRING or FILE_NAME. Type defaults to STRING, except on X11, where the default is to try UTF8_STRING and fall back to STRING.

This command is equivalent to:

selection_get(CLIPBOARD)

colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

colormodel(*value=None*)
Useless. Not implemented in Tk.

columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

config(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

configure(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

deletecommand(*name*)
Internal function.

Delete the Tcl command provided in NAME.

destroy()
Destroy this and all descendants widgets.

event_add(*virtual*, **sequences*)
Bind a virtual event VIRTUAL (of the form <<Name>>) to an event SEQUENCE such that the virtual event is triggered whenever SEQUENCE occurs.

event_delete(*virtual*, **sequences*)
Unbind a virtual event VIRTUAL from SEQUENCE.

event_generate(*sequence*, ***kw*)
Generate an event SEQUENCE. Additional keyword arguments specify parameter of the event (e.g. x, y, rootx, rooty).

event_info(*virtual=None*)
Return a list of all virtual events or the information about the SEQUENCE bound to the virtual event VIRTUAL.

focus()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focus_displayof()
Return the widget which has currently the focus on the display where this widget is located.

Return None if the application does not have the focus.

focus_force()
Direct input focus to this widget even if the application does not have the focus. Use with caution!

focus_get()
Return the widget which has currently the focus in the application.

Use focus_displayof to allow working with several displays. Return None if application does not have the focus.

focus_lastfor()
Return the widget which would have the focus if top level for this widget gets the focus from the window manager.

focus_set()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

frame()
Return identifier for decorative frame of this widget if present.

geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

getboolean(*s*)
Return a boolean value for Tcl boolean values true and false given as parameter.

getdouble
alias of `__builtin__.float`

getint
alias of `__builtin__.int`

getvar(*name='PY_VAR'*)
Return value of Tcl variable NAME.

grab_current()
Return widget which has currently the grab in this application or None.

grab_release()
Release grab for this widget if currently set.

grab_set(*timeout=30*)

grab_set_global()
Set global grab for this widget.

A global grab directs all events to this and descendant widgets on the display. Use with caution - other applications do not get events anymore.

grab_status()
Return None, "local" or "global" if this widget has no, a local or a global grab.

grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

grid_bbox(*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

grid_columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

grid_location(*x*, *y*)
Return a tuple of column and row which identify the cell at which the pixel at position X and Y inside the master widget is located.

grid_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given, the current setting will be returned.

grid_rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

grid_size()
Return a tuple of the number of column and rows in the grid.

grid_slaves(*row=None*, *column=None*)
Return a list of all slaves of this widget in its packing order.

group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

iconify()
Display widget as icon.

iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

image_names()
Return a list of all existing image names.

image_types()
Return a list of all available image types (e.g. photo bitmap).

keys()
Return a list of all resource names of this widget.

lift(*aboveThis=None*)
Raise this widget in the stacking order.

lower(*belowThis=None*)
Lower this widget in the stacking order.

mainloop(*n=0*)
Call the mainloop of Tk.

maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

nametowidget(*name*)
Return the Tkinter instance of a widget identified by its Tcl name NAME.

option_add(*pattern*, *value*, *priority=None*)
Set a VALUE (second parameter) for an option PATTERN (first parameter).

An optional third parameter gives the numeric priority (defaults to 80).

option_clear()
Clear the option database.

It will be reloaded if option_add is called.

option_get(*name*, *className*)
Return the value for an option NAME for this widget with CLASSNAME.

Values with higher priority override lower values.

option_readfile(*fileName*, *priority=None*)
Read file FILENAME into the option database.

An optional second parameter gives the numeric priority.

overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

pack_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

pack_slaves()
Return a list of all slaves of this widget in its packing order.

place_slaves()
Return a list of all slaves of this widget in its packing order.

positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

quit()
Quit the Tcl interpreter. All widgets will be destroyed.

register(*func*, *subst=None*, *needcleanup=1*)
Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be executed before FUNC.

resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

selection_clear(***kw*)
Clear the current X selection.

selection_get(**kw)
Return the contents of the current X selection.

A keyword parameter selection specifies the name of the selection and defaults to PRIMARY. A keyword parameter displayof specifies a widget on the display to use. A keyword parameter type specifies the form of data to be fetched, defaulting to STRING except on X11, where UTF8_STRING is tried before STRING.

selection_handle(*command*, **kw)
Specify a function COMMAND to call if the X selection owned by this widget is queried by another application.

This function must return the contents of the selection. The function will be called with the arguments OFFSET and LENGTH which allows the chunking of very long selections. The following keyword parameters can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

selection_own(**kw)
Become owner of X selection.

A keyword parameter selection specifies the name of the selection (default PRIMARY).

selection_own_get(**kw)
Return owner of X selection.

The following keyword parameter can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

send(*interp*, *cmd*, *args)
Send Tcl command CMD to different interpreter INTERP to be executed.

setvar(*name='PY_VAR'*, *value='1'*)
Set Tcl variable NAME to VALUE.

show()

size()
Return a tuple of the number of column and rows in the grid.

sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

slaves()
Return a list of all slaves of this widget in its packing order.

state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

title(*string=None*)
Set the title of this widget.

tk_bisque()
Change the color scheme to light brown as used in Tk 3.6 and before.

tk_focusFollowsMouse()
The widget under mouse will get automatically focus. Can not be disabled easily.

tk_focusNext()
Return the next widget in the focus order which follows widget which has currently the focus.

The focus order first goes to the next child, then to the children of the child recursively and then to the next sibling which is higher in the stacking order. A widget is omitted if it has the takefocus resource set to 0.

tk_focusPrev()
Return previous widget in the focus order. See tk_focusNext for details.

tk_menuBar(**args*)
Do not use. Needed in Tk 3.6 and earlier.

tk_setPalette(**args*, ***kw*)
Set a new color scheme for all widget elements.

A single color as argument will cause that all colors of Tk widget elements are derived from this. Alternatively several keyword parameters and its associated colors can be given. The following keywords are valid: activeBackground, foreground, selectColor, activeForeground, highlightBackground, selectBackground, background, highlightColor, selectForeground, disabledForeground, insertBackground, troughColor.

tk_strictMotif(*boolean=None*)
Set Tcl internal variable, whether the look and feel should adhere to Motif.

A parameter of 1 means adhere to Motif (e.g. no color change if mouse passes over slider). Returns the set value.

tkraise(*aboveThis=None*)
Raise this widget in the stacking order.

transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

unbind(*sequence*, *funcid=None*)
Unbind for this widget for event SEQUENCE the function identified with FUNCID.

unbind_all(*sequence*)
Unbind for all widgets for event SEQUENCE all functions.

unbind_class(*className*, *sequence*)
Unbind for all widgets with bindtag CLASSNAME for event SEQUENCE all functions.

update()
Enter event loop until all pending events have been processed by Tcl.

update_idletasks()
Enter event loop until all idle callbacks have been called. This will update the display of windows but not process events caused by the user.

wait_variable(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

wait_visibility(*window=None*)
Wait until the visibility of a WIDGET changes (e.g. it appears).

If no parameter is given self is used.

wait_window(*window=None*)
Wait until a WIDGET is destroyed.

If no parameter is given self is used.

waitvar(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

winfo_atom(*name*, *displayof=0*)
Return integer which represents atom NAME.

winfo_atomname(*id*, *displayof=0*)
Return name of atom with identifier ID.

winfo_cells()
Return number of cells in the colormap for this widget.

winfo_children()
Return a list of all widgets which are children of this widget.

winfo_class()
Return window class name of this widget.

winfo_colormapfull()
Return true if at the last color request the colormap was full.

winfo_containing(*rootX*, *rootY*, *displayof=0*)
Return the widget which is at the root coordinates ROOTX, ROOTY.

winfo_depth()
Return the number of bits per pixel.

winfo_exists()
Return true if this widget exists.

winfo_fpixels(*number*)
Return the number of pixels for the given distance NUMBER (e.g. “3c”) as float.

winfo_geometry()
Return geometry string for this widget in the form “widthxheight+X+Y”.

winfo_height()
Return height of this widget.

winfo_id()
Return identifier ID for this widget.

winfo_interps(*displayof=0*)
Return the name of all Tcl interpreters for this display.

winfo_ismapped()
Return true if this widget is mapped.

winfo_manager()
Return the window manager name for this widget.

winfo_name()
Return the name of this widget.

winfo_parent()
Return the name of the parent of this widget.

winfo_pathname(*id*, *displayof=0*)
Return the pathname of the widget given by ID.

winfo_pixels(*number*)
Rounded integer value of winfo_fpixels.

winfo_pointerx()
Return the x coordinate of the pointer on the root window.

winfo_pointerxy()
Return a tuple of x and y coordinates of the pointer on the root window.

winfo_pointery()
Return the y coordinate of the pointer on the root window.

winfo_reqheight()
Return requested height of this widget.

winfo_reqwidth()
Return requested width of this widget.

winfo_rgb(*color*)
Return tuple of decimal values for red, green, blue for COLOR in this widget.

winfo_rootx()
Return x coordinate of upper left corner of this widget on the root window.

winfo_rooty()
Return y coordinate of upper left corner of this widget on the root window.

winfo_screen()
Return the screen name of this widget.

winfo_screencells()
Return the number of the cells in the colormap of the screen of this widget.

winfo_screendepth()
Return the number of bits per pixel of the root window of the screen of this widget.

winfo_screenheight()
Return the number of pixels of the height of the screen of this widget in pixel.

winfo_screenmmheight()
Return the number of pixels of the height of the screen of this widget in mm.

winfo_screenmmwidth()
Return the number of pixels of the width of the screen of this widget in mm.

winfo_screenvisual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the default colormodel of this screen.

winfo_screenwidth()
Return the number of pixels of the width of the screen of this widget in pixel.

winfo_server()
Return information of the X-Server of the screen of this widget in the form "XmajorRminor vendor vendorVersion".

winfo_toplevel()
Return the toplevel widget of this widget.

winfo_viewable()
Return true if the widget and all its higher ancestors are mapped.

winfo_visual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the colormodel of this widget.

winfo_visualid()
Return the X identifier for the visual for this widget.

winfo_visualsavailable(*includeids=0*)
Return a list of all visuals available for the screen of this widget.

Each item in the list consists of a visual name (see winfo_visual), a depth and if INCLUDEIDS=1 is given also the X identifier.

winfo_vrootheight()
Return the height of the virtual root window associated with this widget in pixels. If there is no virtual root window return the height of the screen.

winfo_vrootwidth()
Return the width of the virtual root window associated with this widget in pixel. If there is no virtual root window return the width of the screen.

winfo_vrootx()
Return the x offset of the virtual root relative to the root window of the screen of this widget.

winfo_vrooty()
Return the y offset of the virtual root relative to the root window of the screen of this widget.

winfo_width()
Return the width of this widget.

winfo_x()
Return the x coordinate of the upper left corner of this widget in the parent.

winfo_y()
Return the y coordinate of the upper left corner of this widget in the parent.

withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

wm_aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

wm_attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

wm_client(*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

wm_colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

wm_command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

wm_deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

wm_focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

wm_frame()
Return identifier for decorative frame of this widget if present.

wm_geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

wm_grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

wm_group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

wm_iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

wm_iconify()
Display widget as icon.

wm_iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

wm_iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

wm_iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

wm_iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

wm_maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

wm_positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. “WM_SAVE_YOURSELF” or “WM_DELETE_WINDOW”.

wm_resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

wm_sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is “user”, and by its own policy if WHO is “program”.

wm_state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

wm_title(*string=None*)
Set the title of this widget.

wm_transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

wm_withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

class robot.libraries.dialogs_py.**InputDialog**(*message*, *default=''*, *hidden=False*)
Bases: robot.libraries.dialogs_py._TkDialog

after(*ms*, *func=None*, **args*)
Call function once after given time.

MS specifies the time in milliseconds. FUNC gives the function which shall be called. Additional parameters are given as parameters to the function call. Return identifier to cancel scheduling with after_cancel.

after_cancel(*id*)
Cancel scheduling of function identified with ID.

Identifier returned by after or after_idle must be given as first parameter.

after_idle(*func*, **args*)
Call FUNC once if the Tcl main loop has no event to process.

Return an identifier to cancel the scheduling with after_cancel.

aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

bbox (*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

bell (*displayof=0*)
Ring a display's bell.

bind (*sequence=None*, *func=None*, *add=None*)
Bind to this widget at event SEQUENCE a call to function FUNC.

SEQUENCE is a string of concatenated event patterns. An event pattern is of the form <MODIFIER-MODIFIER-TYPE-DETAIL> where MODIFIER is one of Control, Mod2, M2, Shift, Mod3, M3, Lock, Mod4, M4, Button1, B1, Mod5, M5 Button2, B2, Meta, M, Button3, B3, Alt, Button4, B4, Double, Button5, B5 Triple, Mod1, M1. TYPE is one of Activate, Enter, Map, ButtonPress, Button, Expose, Motion, ButtonRelease FocusIn, MouseWheel, Circulate, FocusOut, Property, Colormap, Gravity Reparent, Configure, KeyPress, Key, Unmap, Deactivate, KeyRelease Visibility, Destroy, Leave and DETAIL is the button number for ButtonPress, ButtonRelease and DETAIL is the Keysym for KeyPress and KeyRelease. Examples are <Control-Button-1> for pressing Control and mouse button 1 or <Alt-A> for pressing A and the Alt key (KeyPress can be omitted). An event pattern can also be a virtual event of the form <<AString>> where AString can be arbitrary. This event can be generated by event_generate. If events are concatenated they must appear shortly after each other.

FUNC will be called if the event sequence occurs with an instance of Event as argument. If the return value of FUNC is "break" no further bound function is invoked.

An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function.

Bind will return an identifier to allow deletion of the bound function with unbind without memory leak.

If FUNC or SEQUENCE is omitted the bound function or list of bound events are returned.

bind_all (*sequence=None*, *func=None*, *add=None*)
Bind to all widgets at an event SEQUENCE a call to function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bind_class (*className*, *sequence=None*, *func=None*, *add=None*)
Bind to widgets with bindtag CLASSNAME at event SEQUENCE a call of function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bindtags (*tagList=None*)
Set or get the list of bindtags for this widget.

With no argument return the list of all bindtags associated with this widget. With a list of strings as argument the bindtags are set to this list. The bindtags determine in which order events are processed (see bind).

cget (*key*)
Return the resource value for a KEY given as string.

client (*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

clipboard_append(*string*, ***kw*)
Append STRING to the Tk clipboard.

A widget specified at the optional displayof keyword argument specifies the target display. The clipboard can be retrieved with selection_get.

clipboard_clear(***kw*)
Clear the data in the Tk clipboard.

A widget specified for the optional displayof keyword argument specifies the target display.

clipboard_get(***kw*)
Retrieve data from the clipboard on window's display.

The window keyword defaults to the root window of the Tkinter application.

The type keyword specifies the form in which the data is to be returned and should be an atom name such as STRING or FILE_NAME. Type defaults to STRING, except on X11, where the default is to try UTF8_STRING and fall back to STRING.

This command is equivalent to:

selection_get(CLIPBOARD)

colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

colormodel(*value=None*)
Useless. Not implemented in Tk.

columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

config(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

configure(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

deletecommand(*name*)
Internal function.

Delete the Tcl command provided in NAME.

destroy()
Destroy this and all descendants widgets.

event_add(*virtual*, **sequences*)
Bind a virtual event VIRTUAL (of the form <<Name>>) to an event SEQUENCE such that the virtual event is triggered whenever SEQUENCE occurs.

event_delete(*virtual*, **sequences*)
Unbind a virtual event VIRTUAL from SEQUENCE.

event_generate(*sequence*, ***kw*)
Generate an event SEQUENCE. Additional keyword arguments specify parameter of the event (e.g. x, y, rootx, rooty).

event_info(*virtual=None*)
Return a list of all virtual events or the information about the SEQUENCE bound to the virtual event VIRTUAL.

focus()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focus_displayof()
Return the widget which has currently the focus on the display where this widget is located.

Return None if the application does not have the focus.

focus_force()
Direct input focus to this widget even if the application does not have the focus. Use with caution!

focus_get()
Return the widget which has currently the focus in the application.

Use focus_displayof to allow working with several displays. Return None if application does not have the focus.

focus_lastfor()
Return the widget which would have the focus if top level for this widget gets the focus from the window manager.

focus_set()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

frame()
Return identifier for decorative frame of this widget if present.

geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

getboolean(*s*)
Return a boolean value for Tcl boolean values true and false given as parameter.

getdouble
alias of `__builtin__.float`

getint
alias of __builtin__.int

getvar(*name='PY_VAR'*)
Return value of Tcl variable NAME.

grab_current()
Return widget which has currently the grab in this application or None.

grab_release()
Release grab for this widget if currently set.

grab_set(*timeout=30*)

grab_set_global()
Set global grab for this widget.

A global grab directs all events to this and descendant widgets on the display. Use with caution - other applications do not get events anymore.

grab_status()
Return None, "local" or "global" if this widget has no, a local or a global grab.

grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

grid_bbox(*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

grid_columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

grid_location(*x*, *y*)
Return a tuple of column and row which identify the cell at which the pixel at position X and Y inside the master widget is located.

grid_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given, the current setting will be returned.

grid_rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

grid_size()
Return a tuple of the number of column and rows in the grid.

grid_slaves(*row=None*, *column=None*)
Return a list of all slaves of this widget in its packing order.

group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

iconify()
Display widget as icon.

iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

image_names()
Return a list of all existing image names.

image_types()
Return a list of all available image types (e.g. photo bitmap).

keys()
Return a list of all resource names of this widget.

lift(*aboveThis=None*)
Raise this widget in the stacking order.

lower(*belowThis=None*)
Lower this widget in the stacking order.

mainloop(*n=0*)
Call the mainloop of Tk.

maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

nametowidget(*name*)
Return the Tkinter instance of a widget identified by its Tcl name NAME.

option_add(*pattern*, *value*, *priority=None*)
Set a VALUE (second parameter) for an option PATTERN (first parameter).

An optional third parameter gives the numeric priority (defaults to 80).

option_clear()
Clear the option database.

It will be reloaded if option_add is called.

option_get(*name*, *className*)
Return the value for an option NAME for this widget with CLASSNAME.

Values with higher priority override lower values.

option_readfile(*fileName*, *priority=None*)
Read file FILENAME into the option database.

An optional second parameter gives the numeric priority.

overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

pack_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

pack_slaves()
Return a list of all slaves of this widget in its packing order.

place_slaves()
Return a list of all slaves of this widget in its packing order.

positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

quit()
Quit the Tcl interpreter. All widgets will be destroyed.

register(*func*, *subst=None*, *needcleanup=1*)
Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be executed before FUNC.

resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

selection_clear(***kw*)
Clear the current X selection.

selection_get(**kw)
Return the contents of the current X selection.

A keyword parameter selection specifies the name of the selection and defaults to PRIMARY. A keyword parameter displayof specifies a widget on the display to use. A keyword parameter type specifies the form of data to be fetched, defaulting to STRING except on X11, where UTF8_STRING is tried before STRING.

selection_handle(*command*, **kw)
Specify a function COMMAND to call if the X selection owned by this widget is queried by another application.

This function must return the contents of the selection. The function will be called with the arguments OFFSET and LENGTH which allows the chunking of very long selections. The following keyword parameters can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

selection_own(**kw)
Become owner of X selection.

A keyword parameter selection specifies the name of the selection (default PRIMARY).

selection_own_get(**kw)
Return owner of X selection.

The following keyword parameter can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

send(*interp*, *cmd*, **args*)
Send Tcl command CMD to different interpreter INTERP to be executed.

setvar(*name='PY_VAR'*, *value='1'*)
Set Tcl variable NAME to VALUE.

show()

size()
Return a tuple of the number of column and rows in the grid.

sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

slaves()
Return a list of all slaves of this widget in its packing order.

state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

title(*string=None*)
Set the title of this widget.

tk_bisque()
Change the color scheme to light brown as used in Tk 3.6 and before.

tk_focusFollowsMouse()
The widget under mouse will get automatically focus. Can not be disabled easily.

tk_focusNext()
Return the next widget in the focus order which follows widget which has currently the focus.

The focus order first goes to the next child, then to the children of the child recursively and then to the next sibling which is higher in the stacking order. A widget is omitted if it has the takefocus resource set to 0.

tk_focusPrev()
Return previous widget in the focus order. See tk_focusNext for details.

tk_menuBar(**args*)
Do not use. Needed in Tk 3.6 and earlier.

tk_setPalette(**args*, ***kw*)
Set a new color scheme for all widget elements.

A single color as argument will cause that all colors of Tk widget elements are derived from this. Alternatively several keyword parameters and its associated colors can be given. The following keywords are valid: activeBackground, foreground, selectColor, activeForeground, highlightBackground, selectBackground, background, highlightColor, selectForeground, disabledForeground, insertBackground, troughColor.

tk_strictMotif(*boolean=None*)
Set Tcl internal variable, whether the look and feel should adhere to Motif.

A parameter of 1 means adhere to Motif (e.g. no color change if mouse passes over slider). Returns the set value.

tkraise(*aboveThis=None*)
Raise this widget in the stacking order.

transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

unbind(*sequence*, *funcid=None*)
Unbind for this widget for event SEQUENCE the function identified with FUNCID.

unbind_all(*sequence*)
Unbind for all widgets for event SEQUENCE all functions.

unbind_class(*className*, *sequence*)
Unbind for all widgets with bindtag CLASSNAME for event SEQUENCE all functions.

update()
Enter event loop until all pending events have been processed by Tcl.

update_idletasks()
Enter event loop until all idle callbacks have been called. This will update the display of windows but not process events caused by the user.

wait_variable(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

wait_visibility(*window=None*)
Wait until the visibility of a WIDGET changes (e.g. it appears).

If no parameter is given self is used.

wait_window(*window=None*)
Wait until a WIDGET is destroyed.

If no parameter is given self is used.

waitvar(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

winfo_atom(*name*, *displayof=0*)
Return integer which represents atom NAME.

winfo_atomname(*id*, *displayof=0*)
Return name of atom with identifier ID.

winfo_cells()
Return number of cells in the colormap for this widget.

winfo_children()
Return a list of all widgets which are children of this widget.

winfo_class()
Return window class name of this widget.

winfo_colormapfull()
Return true if at the last color request the colormap was full.

winfo_containing(*rootX*, *rootY*, *displayof=0*)
Return the widget which is at the root coordinates ROOTX, ROOTY.

winfo_depth()
Return the number of bits per pixel.

winfo_exists()
Return true if this widget exists.

winfo_fpixels(*number*)
Return the number of pixels for the given distance NUMBER (e.g. “3c”) as float.

winfo_geometry()
Return geometry string for this widget in the form “widthxheight+X+Y”.

winfo_height()
Return height of this widget.

winfo_id()
Return identifier ID for this widget.

winfo_interps(*displayof=0*)
Return the name of all Tcl interpreters for this display.

winfo_ismapped()
Return true if this widget is mapped.

winfo_manager()
Return the window manager name for this widget.

winfo_name()
Return the name of this widget.

winfo_parent()
Return the name of the parent of this widget.

winfo_pathname(*id*, *displayof=0*)
Return the pathname of the widget given by ID.

winfo_pixels(*number*)
Rounded integer value of winfo_fpixels.

winfo_pointerx()
Return the x coordinate of the pointer on the root window.

winfo_pointerxy()
Return a tuple of x and y coordinates of the pointer on the root window.

winfo_pointery()
Return the y coordinate of the pointer on the root window.

winfo_reqheight()
Return requested height of this widget.

winfo_reqwidth()
Return requested width of this widget.

winfo_rgb(*color*)
Return tuple of decimal values for red, green, blue for COLOR in this widget.

winfo_rootx()
Return x coordinate of upper left corner of this widget on the root window.

winfo_rooty()
Return y coordinate of upper left corner of this widget on the root window.

winfo_screen()
Return the screen name of this widget.

winfo_screencells()
Return the number of the cells in the colormap of the screen of this widget.

winfo_screendepth()
Return the number of bits per pixel of the root window of the screen of this widget.

winfo_screenheight()
Return the number of pixels of the height of the screen of this widget in pixel.

winfo_screenmmheight()
Return the number of pixels of the height of the screen of this widget in mm.

winfo_screenmmwidth()
Return the number of pixels of the width of the screen of this widget in mm.

winfo_screenvisual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the default colormodel of this screen.

winfo_screenwidth()
Return the number of pixels of the width of the screen of this widget in pixel.

winfo_server()
Return information of the X-Server of the screen of this widget in the form "XmajorRminor vendor vendorVersion".

winfo_toplevel()
Return the toplevel widget of this widget.

winfo_viewable()
Return true if the widget and all its higher ancestors are mapped.

winfo_visual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the colormodel of this widget.

winfo_visualid()
Return the X identifier for the visual for this widget.

winfo_visualsavailable(*includeids=0*)
Return a list of all visuals available for the screen of this widget.

Each item in the list consists of a visual name (see winfo_visual), a depth and if INCLUDEIDS=1 is given also the X identifier.

winfo_vrootheight()
Return the height of the virtual root window associated with this widget in pixels. If there is no virtual root window return the height of the screen.

winfo_vrootwidth()
Return the width of the virtual root window associated with this widget in pixel. If there is no virtual root window return the width of the screen.

winfo_vrootx()
Return the x offset of the virtual root relative to the root window of the screen of this widget.

winfo_vrooty()
Return the y offset of the virtual root relative to the root window of the screen of this widget.

winfo_width()
Return the width of this widget.

winfo_x()
Return the x coordinate of the upper left corner of this widget in the parent.

winfo_y()
Return the y coordinate of the upper left corner of this widget in the parent.

withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

wm_aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

wm_attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

wm_client(*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

wm_colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

wm_command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

wm_deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

wm_focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

wm_frame()
Return identifier for decorative frame of this widget if present.

wm_geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

wm_grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

wm_group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

wm_iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

wm_iconify()
Display widget as icon.

wm_iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

wm_iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

wm_iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

wm_iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

wm_maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

wm_positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

wm_resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

wm_sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

wm_title(*string=None*)
Set the title of this widget.

wm_transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

wm_withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

class robot.libraries.dialogs_py.**SelectionDialog**(*message*, *values*)
Bases: robot.libraries.dialogs_py._TkDialog

after(*ms*, *func=None*, **args*)
Call function once after given time.

MS specifies the time in milliseconds. FUNC gives the function which shall be called. Additional parameters are given as parameters to the function call. Return identifier to cancel scheduling with after_cancel.

after_cancel(*id*)
Cancel scheduling of function identified with ID.

Identifier returned by after or after_idle must be given as first parameter.

after_idle(*func*, **args*)
Call FUNC once if the Tcl main loop has no event to process.

Return an identifier to cancel the scheduling with after_cancel.

aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

bbox (*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

bell (*displayof=0*)
Ring a display's bell.

bind (*sequence=None*, *func=None*, *add=None*)
Bind to this widget at event SEQUENCE a call to function FUNC.

SEQUENCE is a string of concatenated event patterns. An event pattern is of the form <MODIFIER-MODIFIER-TYPE-DETAIL> where MODIFIER is one of Control, Mod2, M2, Shift, Mod3, M3, Lock, Mod4, M4, Button1, B1, Mod5, M5 Button2, B2, Meta, M, Button3, B3, Alt, Button4, B4, Double, Button5, B5 Triple, Mod1, M1. TYPE is one of Activate, Enter, Map, ButtonPress, Button, Expose, Motion, ButtonRelease FocusIn, MouseWheel, Circulate, FocusOut, Property, Colormap, Gravity Reparent, Configure, KeyPress, Key, Unmap, Deactivate, KeyRelease Visibility, Destroy, Leave and DETAIL is the button number for ButtonPress, ButtonRelease and DETAIL is the Keysym for KeyPress and KeyRelease. Examples are <Control-Button-1> for pressing Control and mouse button 1 or <Alt-A> for pressing A and the Alt key (KeyPress can be omitted). An event pattern can also be a virtual event of the form <<AString>> where AString can be arbitrary. This event can be generated by event_generate. If events are concatenated they must appear shortly after each other.

FUNC will be called if the event sequence occurs with an instance of Event as argument. If the return value of FUNC is "break" no further bound function is invoked.

An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function.

Bind will return an identifier to allow deletion of the bound function with unbind without memory leak.

If FUNC or SEQUENCE is omitted the bound function or list of bound events are returned.

bind_all (*sequence=None*, *func=None*, *add=None*)
Bind to all widgets at an event SEQUENCE a call to function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bind_class (*className*, *sequence=None*, *func=None*, *add=None*)
Bind to widgets with bindtag CLASSNAME at event SEQUENCE a call of function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bindtags (*tagList=None*)
Set or get the list of bindtags for this widget.

With no argument return the list of all bindtags associated with this widget. With a list of strings as argument the bindtags are set to this list. The bindtags determine in which order events are processed (see bind).

cget (*key*)
Return the resource value for a KEY given as string.

client (*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

clipboard_append(*string*, ***kw*)
Append STRING to the Tk clipboard.

A widget specified at the optional displayof keyword argument specifies the target display. The clipboard can be retrieved with selection_get.

clipboard_clear(***kw*)
Clear the data in the Tk clipboard.

A widget specified for the optional displayof keyword argument specifies the target display.

clipboard_get(***kw*)
Retrieve data from the clipboard on window's display.

The window keyword defaults to the root window of the Tkinter application.

The type keyword specifies the form in which the data is to be returned and should be an atom name such as STRING or FILE_NAME. Type defaults to STRING, except on X11, where the default is to try UTF8_STRING and fall back to STRING.

This command is equivalent to:

selection_get(CLIPBOARD)

colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

colormodel(*value=None*)
Useless. Not implemented in Tk.

columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

config(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

configure(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

deletecommand(*name*)
Internal function.

Delete the Tcl command provided in NAME.

destroy()
Destroy this and all descendants widgets.

event_add(*virtual*, **sequences*)
Bind a virtual event VIRTUAL (of the form <<Name>>) to an event SEQUENCE such that the virtual event is triggered whenever SEQUENCE occurs.

event_delete(*virtual*, **sequences*)
Unbind a virtual event VIRTUAL from SEQUENCE.

event_generate(*sequence*, ***kw*)
Generate an event SEQUENCE. Additional keyword arguments specify parameter of the event (e.g. x, y, rootx, rooty).

event_info(*virtual=None*)
Return a list of all virtual events or the information about the SEQUENCE bound to the virtual event VIRTUAL.

focus()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focus_displayof()
Return the widget which has currently the focus on the display where this widget is located.

Return None if the application does not have the focus.

focus_force()
Direct input focus to this widget even if the application does not have the focus. Use with caution!

focus_get()
Return the widget which has currently the focus in the application.

Use focus_displayof to allow working with several displays. Return None if application does not have the focus.

focus_lastfor()
Return the widget which would have the focus if top level for this widget gets the focus from the window manager.

focus_set()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

frame()
Return identifier for decorative frame of this widget if present.

geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

getboolean(*s*)
Return a boolean value for Tcl boolean values true and false given as parameter.

getdouble
alias of `__builtin__.float`

getint
alias of `__builtin__.int`

getvar(*name='PY_VAR'*)
Return value of Tcl variable NAME.

grab_current()
Return widget which has currently the grab in this application or None.

grab_release()
Release grab for this widget if currently set.

grab_set(*timeout=30*)

grab_set_global()
Set global grab for this widget.

A global grab directs all events to this and descendant widgets on the display. Use with caution - other applications do not get events anymore.

grab_status()
Return None, "local" or "global" if this widget has no, a local or a global grab.

grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

grid_bbox(*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

grid_columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

grid_location(*x*, *y*)
Return a tuple of column and row which identify the cell at which the pixel at position X and Y inside the master widget is located.

grid_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given, the current setting will be returned.

grid_rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

grid_size()
Return a tuple of the number of column and rows in the grid.

grid_slaves(*row=None*, *column=None*)
Return a list of all slaves of this widget in its packing order.

group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

iconify()
Display widget as icon.

iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

image_names()
Return a list of all existing image names.

image_types()
Return a list of all available image types (e.g. photo bitmap).

keys()
Return a list of all resource names of this widget.

lift(*aboveThis=None*)
Raise this widget in the stacking order.

lower(*belowThis=None*)
Lower this widget in the stacking order.

mainloop(*n=0*)
Call the mainloop of Tk.

maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

nametowidget(*name*)
Return the Tkinter instance of a widget identified by its Tcl name NAME.

option_add(*pattern*, *value*, *priority=None*)
Set a VALUE (second parameter) for an option PATTERN (first parameter).

An optional third parameter gives the numeric priority (defaults to 80).

option_clear()
Clear the option database.

It will be reloaded if option_add is called.

option_get(*name*, *className*)
Return the value for an option NAME for this widget with CLASSNAME.

Values with higher priority override lower values.

option_readfile(*fileName*, *priority=None*)
Read file FILENAME into the option database.

An optional second parameter gives the numeric priority.

overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

pack_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

pack_slaves()
Return a list of all slaves of this widget in its packing order.

place_slaves()
Return a list of all slaves of this widget in its packing order.

positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

quit()
Quit the Tcl interpreter. All widgets will be destroyed.

register(*func*, *subst=None*, *needcleanup=1*)
Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be executed before FUNC.

resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

selection_clear(***kw*)
Clear the current X selection.

selection_get(**kw)
Return the contents of the current X selection.

A keyword parameter selection specifies the name of the selection and defaults to PRIMARY. A keyword parameter displayof specifies a widget on the display to use. A keyword parameter type specifies the form of data to be fetched, defaulting to STRING except on X11, where UTF8_STRING is tried before STRING.

selection_handle(*command*, **kw)
Specify a function COMMAND to call if the X selection owned by this widget is queried by another application.

This function must return the contents of the selection. The function will be called with the arguments OFFSET and LENGTH which allows the chunking of very long selections. The following keyword parameters can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

selection_own(**kw)
Become owner of X selection.

A keyword parameter selection specifies the name of the selection (default PRIMARY).

selection_own_get(**kw)
Return owner of X selection.

The following keyword parameter can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

send(*interp*, *cmd*, **args*)
Send Tcl command CMD to different interpreter INTERP to be executed.

setvar(*name='PY_VAR'*, *value='1'*)
Set Tcl variable NAME to VALUE.

show()

size()
Return a tuple of the number of column and rows in the grid.

sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

slaves()
Return a list of all slaves of this widget in its packing order.

state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

title(*string=None*)
Set the title of this widget.

tk_bisque()
Change the color scheme to light brown as used in Tk 3.6 and before.

tk_focusFollowsMouse()
The widget under mouse will get automatically focus. Can not be disabled easily.

tk_focusNext()
Return the next widget in the focus order which follows widget which has currently the focus.

The focus order first goes to the next child, then to the children of the child recursively and then to the next sibling which is higher in the stacking order. A widget is omitted if it has the takefocus resource set to 0.

tk_focusPrev()
Return previous widget in the focus order. See tk_focusNext for details.

tk_menuBar(**args*)
Do not use. Needed in Tk 3.6 and earlier.

tk_setPalette(**args*, ***kw*)
Set a new color scheme for all widget elements.

A single color as argument will cause that all colors of Tk widget elements are derived from this. Alternatively several keyword parameters and its associated colors can be given. The following keywords are valid: activeBackground, foreground, selectColor, activeForeground, highlightBackground, selectBackground, background, highlightColor, selectForeground, disabledForeground, insertBackground, troughColor.

tk_strictMotif(*boolean=None*)
Set Tcl internal variable, whether the look and feel should adhere to Motif.

A parameter of 1 means adhere to Motif (e.g. no color change if mouse passes over slider). Returns the set value.

tkraise(*aboveThis=None*)
Raise this widget in the stacking order.

transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

unbind(*sequence*, *funcid=None*)
Unbind for this widget for event SEQUENCE the function identified with FUNCID.

unbind_all(*sequence*)
Unbind for all widgets for event SEQUENCE all functions.

unbind_class(*className*, *sequence*)
Unbind for all widgets with bindtag CLASSNAME for event SEQUENCE all functions.

update()
Enter event loop until all pending events have been processed by Tcl.

update_idletasks()
Enter event loop until all idle callbacks have been called. This will update the display of windows but not process events caused by the user.

wait_variable(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

wait_visibility(*window=None*)
Wait until the visibility of a WIDGET changes (e.g. it appears).

If no parameter is given self is used.

wait_window(*window=None*)
Wait until a WIDGET is destroyed.

If no parameter is given self is used.

waitvar(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

winfo_atom(*name*, *displayof=0*)
Return integer which represents atom NAME.

winfo_atomname(*id*, *displayof=0*)
Return name of atom with identifier ID.

winfo_cells()
Return number of cells in the colormap for this widget.

winfo_children()
Return a list of all widgets which are children of this widget.

winfo_class()
Return window class name of this widget.

winfo_colormapfull()
Return true if at the last color request the colormap was full.

winfo_containing(*rootX*, *rootY*, *displayof=0*)
Return the widget which is at the root coordinates ROOTX, ROOTY.

winfo_depth()
Return the number of bits per pixel.

winfo_exists()
Return true if this widget exists.

winfo_fpixels(*number*)
Return the number of pixels for the given distance NUMBER (e.g. “3c”) as float.

winfo_geometry()
Return geometry string for this widget in the form “widthxheight+X+Y”.

winfo_height()
Return height of this widget.

winfo_id()
Return identifier ID for this widget.

winfo_interps(*displayof=0*)
Return the name of all Tcl interpreters for this display.

winfo_ismapped()
Return true if this widget is mapped.

winfo_manager()
Return the window manager name for this widget.

winfo_name()
Return the name of this widget.

winfo_parent()
Return the name of the parent of this widget.

winfo_pathname(*id*, *displayof=0*)
Return the pathname of the widget given by ID.

winfo_pixels(*number*)
Rounded integer value of winfo_fpixels.

winfo_pointerx()
Return the x coordinate of the pointer on the root window.

winfo_pointerxy()
Return a tuple of x and y coordinates of the pointer on the root window.

winfo_pointery()
Return the y coordinate of the pointer on the root window.

winfo_reqheight()
Return requested height of this widget.

winfo_reqwidth()
Return requested width of this widget.

winfo_rgb(*color*)
Return tuple of decimal values for red, green, blue for COLOR in this widget.

winfo_rootx()
Return x coordinate of upper left corner of this widget on the root window.

winfo_rooty()
Return y coordinate of upper left corner of this widget on the root window.

winfo_screen()
Return the screen name of this widget.

winfo_screencells()
Return the number of the cells in the colormap of the screen of this widget.

winfo_screendepth()
Return the number of bits per pixel of the root window of the screen of this widget.

winfo_screenheight()
Return the number of pixels of the height of the screen of this widget in pixel.

winfo_screenmmheight()
Return the number of pixels of the height of the screen of this widget in mm.

winfo_screenmmwidth()
Return the number of pixels of the width of the screen of this widget in mm.

winfo_screenvisual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the default colormodel of this screen.

winfo_screenwidth()
Return the number of pixels of the width of the screen of this widget in pixel.

winfo_server()
Return information of the X-Server of the screen of this widget in the form "XmajorRminor vendor vendorVersion".

winfo_toplevel()
Return the toplevel widget of this widget.

winfo_viewable()
Return true if the widget and all its higher ancestors are mapped.

winfo_visual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the colormodel of this widget.

winfo_visualid()
Return the X identifier for the visual for this widget.

winfo_visualsavailable(*includeids=0*)
Return a list of all visuals available for the screen of this widget.

Each item in the list consists of a visual name (see winfo_visual), a depth and if INCLUDEIDS=1 is given also the X identifier.

winfo_vrootheight()
Return the height of the virtual root window associated with this widget in pixels. If there is no virtual root window return the height of the screen.

winfo_vrootwidth()
Return the width of the virtual root window associated with this widget in pixel. If there is no virtual root window return the width of the screen.

winfo_vrootx()
Return the x offset of the virtual root relative to the root window of the screen of this widget.

winfo_vrooty()
Return the y offset of the virtual root relative to the root window of the screen of this widget.

winfo_width()
Return the width of this widget.

winfo_x()
Return the x coordinate of the upper left corner of this widget in the parent.

winfo_y()
Return the y coordinate of the upper left corner of this widget in the parent.

withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

wm_aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

wm_attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

wm_client(*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

wm_colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

wm_command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

wm_deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

wm_focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

wm_frame()
Return identifier for decorative frame of this widget if present.

wm_geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

wm_grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

wm_group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

wm_iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

wm_iconify()
Display widget as icon.

wm_iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

wm_iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

wm_iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

wm_iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

wm_maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

wm_positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. “WM_SAVE_YOURSELF” or “WM_DELETE_WINDOW”.

wm_resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

wm_sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is “user”, and by its own policy if WHO is “program”.

wm_state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

wm_title(*string=None*)
Set the title of this widget.

wm_transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

wm_withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

class robot.libraries.dialogs_py.**MultipleSelectionDialog**(*message*, *values*)
Bases: robot.libraries.dialogs_py._TkDialog

after(*ms*, *func=None*, **args*)
Call function once after given time.

MS specifies the time in milliseconds. FUNC gives the function which shall be called. Additional parameters are given as parameters to the function call. Return identifier to cancel scheduling with after_cancel.

after_cancel(*id*)
Cancel scheduling of function identified with ID.

Identifier returned by after or after_idle must be given as first parameter.

after_idle(*func*, **args*)
Call FUNC once if the Tcl main loop has no event to process.

Return an identifier to cancel the scheduling with after_cancel.

aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

bbox (*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

bell (*displayof=0*)
Ring a display's bell.

bind (*sequence=None*, *func=None*, *add=None*)
Bind to this widget at event SEQUENCE a call to function FUNC.

SEQUENCE is a string of concatenated event patterns. An event pattern is of the form <MODIFIER-MODIFIER-TYPE-DETAIL> where MODIFIER is one of Control, Mod2, M2, Shift, Mod3, M3, Lock, Mod4, M4, Button1, B1, Mod5, M5 Button2, B2, Meta, M, Button3, B3, Alt, Button4, B4, Double, Button5, B5 Triple, Mod1, M1. TYPE is one of Activate, Enter, Map, ButtonPress, Button, Expose, Motion, ButtonRelease FocusIn, MouseWheel, Circulate, FocusOut, Property, Colormap, Gravity Reparent, Configure, KeyPress, Key, Unmap, Deactivate, KeyRelease Visibility, Destroy, Leave and DETAIL is the button number for ButtonPress, ButtonRelease and DETAIL is the Keysym for KeyPress and KeyRelease. Examples are <Control-Button-1> for pressing Control and mouse button 1 or <Alt-A> for pressing A and the Alt key (KeyPress can be omitted). An event pattern can also be a virtual event of the form <<AString>> where AString can be arbitrary. This event can be generated by event_generate. If events are concatenated they must appear shortly after each other.

FUNC will be called if the event sequence occurs with an instance of Event as argument. If the return value of FUNC is "break" no further bound function is invoked.

An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function.

Bind will return an identifier to allow deletion of the bound function with unbind without memory leak.

If FUNC or SEQUENCE is omitted the bound function or list of bound events are returned.

bind_all (*sequence=None*, *func=None*, *add=None*)
Bind to all widgets at an event SEQUENCE a call to function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bind_class (*className*, *sequence=None*, *func=None*, *add=None*)
Bind to widgets with bindtag CLASSNAME at event SEQUENCE a call of function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bindtags (*tagList=None*)
Set or get the list of bindtags for this widget.

With no argument return the list of all bindtags associated with this widget. With a list of strings as argument the bindtags are set to this list. The bindtags determine in which order events are processed (see bind).

cget (*key*)
Return the resource value for a KEY given as string.

client (*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

clipboard_append(*string*, ***kw*)
Append STRING to the Tk clipboard.

A widget specified at the optional displayof keyword argument specifies the target display. The clipboard can be retrieved with selection_get.

clipboard_clear(***kw*)
Clear the data in the Tk clipboard.

A widget specified for the optional displayof keyword argument specifies the target display.

clipboard_get(***kw*)
Retrieve data from the clipboard on window's display.

The window keyword defaults to the root window of the Tkinter application.

The type keyword specifies the form in which the data is to be returned and should be an atom name such as STRING or FILE_NAME. Type defaults to STRING, except on X11, where the default is to try UTF8_STRING and fall back to STRING.

This command is equivalent to:

selection_get(CLIPBOARD)

colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

colormodel(*value=None*)
Useless. Not implemented in Tk.

columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

config(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

configure(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

deletecommand(*name*)
Internal function.

Delete the Tcl command provided in NAME.

destroy()
Destroy this and all descendants widgets.

event_add(*virtual*, **sequences*)
Bind a virtual event VIRTUAL (of the form <<Name>>) to an event SEQUENCE such that the virtual event is triggered whenever SEQUENCE occurs.

event_delete(*virtual*, **sequences*)
Unbind a virtual event VIRTUAL from SEQUENCE.

event_generate(*sequence*, ***kw*)
Generate an event SEQUENCE. Additional keyword arguments specify parameter of the event (e.g. x, y, rootx, rooty).

event_info(*virtual=None*)
Return a list of all virtual events or the information about the SEQUENCE bound to the virtual event VIRTUAL.

focus()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focus_displayof()
Return the widget which has currently the focus on the display where this widget is located.

Return None if the application does not have the focus.

focus_force()
Direct input focus to this widget even if the application does not have the focus. Use with caution!

focus_get()
Return the widget which has currently the focus in the application.

Use focus_displayof to allow working with several displays. Return None if application does not have the focus.

focus_lastfor()
Return the widget which would have the focus if top level for this widget gets the focus from the window manager.

focus_set()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

frame()
Return identifier for decorative frame of this widget if present.

geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

getboolean(*s*)
Return a boolean value for Tcl boolean values true and false given as parameter.

getdouble
alias of `__builtin__.float`

getint
alias of `__builtin__.int`

getvar(*name='PY_VAR'*)
Return value of Tcl variable NAME.

grab_current()
Return widget which has currently the grab in this application or None.

grab_release()
Release grab for this widget if currently set.

grab_set(*timeout=30*)

grab_set_global()
Set global grab for this widget.

A global grab directs all events to this and descendant widgets on the display. Use with caution - other applications do not get events anymore.

grab_status()
Return None, "local" or "global" if this widget has no, a local or a global grab.

grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

grid_bbox(*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

grid_columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

grid_location(*x*, *y*)
Return a tuple of column and row which identify the cell at which the pixel at position X and Y inside the master widget is located.

grid_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given, the current setting will be returned.

grid_rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

grid_size()
Return a tuple of the number of column and rows in the grid.

grid_slaves(*row=None*, *column=None*)
Return a list of all slaves of this widget in its packing order.

group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

iconify()
Display widget as icon.

iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

image_names()
Return a list of all existing image names.

image_types()
Return a list of all available image types (e.g. photo bitmap).

keys()
Return a list of all resource names of this widget.

lift(*aboveThis=None*)
Raise this widget in the stacking order.

lower(*belowThis=None*)
Lower this widget in the stacking order.

mainloop(*n=0*)
Call the mainloop of Tk.

maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

nametowidget(*name*)
Return the Tkinter instance of a widget identified by its Tcl name NAME.

option_add(*pattern*, *value*, *priority=None*)
Set a VALUE (second parameter) for an option PATTERN (first parameter).

An optional third parameter gives the numeric priority (defaults to 80).

option_clear()
Clear the option database.

It will be reloaded if option_add is called.

option_get(*name*, *className*)
Return the value for an option NAME for this widget with CLASSNAME.

Values with higher priority override lower values.

option_readfile(*fileName*, *priority=None*)
Read file FILENAME into the option database.

An optional second parameter gives the numeric priority.

overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

pack_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

pack_slaves()
Return a list of all slaves of this widget in its packing order.

place_slaves()
Return a list of all slaves of this widget in its packing order.

positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

quit()
Quit the Tcl interpreter. All widgets will be destroyed.

register(*func*, *subst=None*, *needcleanup=1*)
Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be executed before FUNC.

resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

selection_clear(***kw*)
Clear the current X selection.

selection_get(***kw*)
Return the contents of the current X selection.

A keyword parameter selection specifies the name of the selection and defaults to PRIMARY. A keyword parameter displayof specifies a widget on the display to use. A keyword parameter type specifies the form of data to be fetched, defaulting to STRING except on X11, where UTF8_STRING is tried before STRING.

selection_handle(*command*, ***kw*)
Specify a function COMMAND to call if the X selection owned by this widget is queried by another application.

This function must return the contents of the selection. The function will be called with the arguments OFFSET and LENGTH which allows the chunking of very long selections. The following keyword parameters can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

selection_own(***kw*)
Become owner of X selection.

A keyword parameter selection specifies the name of the selection (default PRIMARY).

selection_own_get(***kw*)
Return owner of X selection.

The following keyword parameter can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

send(*interp*, *cmd*, **args*)
Send Tcl command CMD to different interpreter INTERP to be executed.

setvar(*name='PY_VAR'*, *value='1'*)
Set Tcl variable NAME to VALUE.

show()

size()
Return a tuple of the number of column and rows in the grid.

sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

slaves()
Return a list of all slaves of this widget in its packing order.

state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

title(*string=None*)
Set the title of this widget.

tk_bisque()
Change the color scheme to light brown as used in Tk 3.6 and before.

tk_focusFollowsMouse()
The widget under mouse will get automatically focus. Can not be disabled easily.

tk_focusNext()
Return the next widget in the focus order which follows widget which has currently the focus.

The focus order first goes to the next child, then to the children of the child recursively and then to the next sibling which is higher in the stacking order. A widget is omitted if it has the takefocus resource set to 0.

tk_focusPrev()
Return previous widget in the focus order. See tk_focusNext for details.

tk_menuBar(**args*)
Do not use. Needed in Tk 3.6 and earlier.

tk_setPalette(**args*, ***kw*)
Set a new color scheme for all widget elements.

A single color as argument will cause that all colors of Tk widget elements are derived from this. Alternatively several keyword parameters and its associated colors can be given. The following keywords are valid: activeBackground, foreground, selectColor, activeForeground, highlightBackground, selectBackground, background, highlightColor, selectForeground, disabledForeground, insertBackground, troughColor.

tk_strictMotif(*boolean=None*)
Set Tcl internal variable, whether the look and feel should adhere to Motif.

A parameter of 1 means adhere to Motif (e.g. no color change if mouse passes over slider). Returns the set value.

tkraise(*aboveThis=None*)
Raise this widget in the stacking order.

transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

unbind(*sequence*, *funcid=None*)
Unbind for this widget for event SEQUENCE the function identified with FUNCID.

unbind_all(*sequence*)
Unbind for all widgets for event SEQUENCE all functions.

unbind_class(*className*, *sequence*)
Unbind for all widgets with bindtag CLASSNAME for event SEQUENCE all functions.

update()
Enter event loop until all pending events have been processed by Tcl.

update_idletasks()
Enter event loop until all idle callbacks have been called. This will update the display of windows but not process events caused by the user.

wait_variable(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

wait_visibility(*window=None*)
Wait until the visibility of a WIDGET changes (e.g. it appears).

If no parameter is given self is used.

wait_window(*window=None*)
Wait until a WIDGET is destroyed.

If no parameter is given self is used.

waitvar(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

winfo_atom(*name*, *displayof=0*)
Return integer which represents atom NAME.

winfo_atomname(*id*, *displayof=0*)
Return name of atom with identifier ID.

winfo_cells()
Return number of cells in the colormap for this widget.

winfo_children()
Return a list of all widgets which are children of this widget.

winfo_class()
Return window class name of this widget.

winfo_colormapfull()
Return true if at the last color request the colormap was full.

winfo_containing(*rootX*, *rootY*, *displayof=0*)
Return the widget which is at the root coordinates ROOTX, ROOTY.

winfo_depth()
Return the number of bits per pixel.

winfo_exists()
Return true if this widget exists.

winfo_fpixels(*number*)
Return the number of pixels for the given distance NUMBER (e.g. “3c”) as float.

winfo_geometry()
Return geometry string for this widget in the form “widthxheight+X+Y”.

winfo_height()
Return height of this widget.

winfo_id()
Return identifier ID for this widget.

winfo_interps(*displayof=0*)
Return the name of all Tcl interpreters for this display.

winfo_ismapped()
Return true if this widget is mapped.

winfo_manager()
Return the window manager name for this widget.

winfo_name()
Return the name of this widget.

winfo_parent()
Return the name of the parent of this widget.

winfo_pathname(*id*, *displayof=0*)
Return the pathname of the widget given by ID.

winfo_pixels(*number*)
Rounded integer value of winfo_fpixels.

winfo_pointerx()
Return the x coordinate of the pointer on the root window.

winfo_pointerxy()
Return a tuple of x and y coordinates of the pointer on the root window.

winfo_pointery()
Return the y coordinate of the pointer on the root window.

winfo_reqheight()
Return requested height of this widget.

winfo_reqwidth()
Return requested width of this widget.

winfo_rgb(*color*)
Return tuple of decimal values for red, green, blue for COLOR in this widget.

winfo_rootx()
Return x coordinate of upper left corner of this widget on the root window.

winfo_rooty()
Return y coordinate of upper left corner of this widget on the root window.

winfo_screen()
Return the screen name of this widget.

winfo_screencells()
Return the number of the cells in the colormap of the screen of this widget.

winfo_screendepth()
Return the number of bits per pixel of the root window of the screen of this widget.

winfo_screenheight()
Return the number of pixels of the height of the screen of this widget in pixel.

winfo_screenmmheight()
Return the number of pixels of the height of the screen of this widget in mm.

winfo_screenmmwidth()
Return the number of pixels of the width of the screen of this widget in mm.

winfo_screenvisual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the default colormodel of this screen.

winfo_screenwidth()
Return the number of pixels of the width of the screen of this widget in pixel.

winfo_server()
Return information of the X-Server of the screen of this widget in the form "XmajorRminor vendor vendorVersion".

winfo_toplevel()
Return the toplevel widget of this widget.

winfo_viewable()
Return true if the widget and all its higher ancestors are mapped.

winfo_visual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the colormodel of this widget.

winfo_visualid()
Return the X identifier for the visual for this widget.

winfo_visualsavailable(*includeids=0*)
Return a list of all visuals available for the screen of this widget.

Each item in the list consists of a visual name (see winfo_visual), a depth and if INCLUDEIDS=1 is given also the X identifier.

winfo_vrootheight()
Return the height of the virtual root window associated with this widget in pixels. If there is no virtual root window return the height of the screen.

winfo_vrootwidth()
Return the width of the virtual root window associated with this widget in pixel. If there is no virtual root window return the width of the screen.

winfo_vrootx()
Return the x offset of the virtual root relative to the root window of the screen of this widget.

winfo_vrooty()
Return the y offset of the virtual root relative to the root window of the screen of this widget.

winfo_width()
Return the width of this widget.

winfo_x()
Return the x coordinate of the upper left corner of this widget in the parent.

winfo_y()
Return the y coordinate of the upper left corner of this widget in the parent.

withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

wm_aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

wm_attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

wm_client(*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

wm_colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

wm_command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

wm_deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

wm_focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

wm_frame()
Return identifier for decorative frame of this widget if present.

wm_geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

wm_grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

wm_group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

wm_iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

wm_iconify()
Display widget as icon.

wm_iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

wm_iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

wm_iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

wm_iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

wm_maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

wm_positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

wm_resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

wm_sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

wm_title(*string=None*)
Set the title of this widget.

wm_transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

wm_withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

class robot.libraries.dialogs_py.**PassFailDialog**(*message*, *value=None*, ***extra*)
Bases: robot.libraries.dialogs_py._TkDialog

after(*ms*, *func=None*, **args*)
Call function once after given time.

MS specifies the time in milliseconds. FUNC gives the function which shall be called. Additional parameters are given as parameters to the function call. Return identifier to cancel scheduling with after_cancel.

after_cancel(*id*)
Cancel scheduling of function identified with ID.

Identifier returned by after or after_idle must be given as first parameter.

after_idle(*func*, **args*)
Call FUNC once if the Tcl main loop has no event to process.

Return an identifier to cancel the scheduling with after_cancel.

aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

bbox (*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

bell (*displayof=0*)
Ring a display's bell.

bind (*sequence=None*, *func=None*, *add=None*)
Bind to this widget at event SEQUENCE a call to function FUNC.

SEQUENCE is a string of concatenated event patterns. An event pattern is of the form <MODIFIER-MODIFIER-TYPE-DETAIL> where MODIFIER is one of Control, Mod2, M2, Shift, Mod3, M3, Lock, Mod4, M4, Button1, B1, Mod5, M5 Button2, B2, Meta, M, Button3, B3, Alt, Button4, B4, Double, Button5, B5 Triple, Mod1, M1. TYPE is one of Activate, Enter, Map, ButtonPress, Button, Expose, Motion, ButtonRelease FocusIn, MouseWheel, Circulate, FocusOut, Property, Colormap, Gravity Reparent, Configure, KeyPress, Key, Unmap, Deactivate, KeyRelease Visibility, Destroy, Leave and DETAIL is the button number for ButtonPress, ButtonRelease and DETAIL is the Keysym for KeyPress and KeyRelease. Examples are <Control-Button-1> for pressing Control and mouse button 1 or <Alt-A> for pressing A and the Alt key (KeyPress can be omitted). An event pattern can also be a virtual event of the form <<AString>> where AString can be arbitrary. This event can be generated by event_generate. If events are concatenated they must appear shortly after each other.

FUNC will be called if the event sequence occurs with an instance of Event as argument. If the return value of FUNC is "break" no further bound function is invoked.

An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function.

Bind will return an identifier to allow deletion of the bound function with unbind without memory leak.

If FUNC or SEQUENCE is omitted the bound function or list of bound events are returned.

bind_all (*sequence=None*, *func=None*, *add=None*)
Bind to all widgets at an event SEQUENCE a call to function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bind_class (*className*, *sequence=None*, *func=None*, *add=None*)
Bind to widgets with bindtag CLASSNAME at event SEQUENCE a call of function FUNC. An additional boolean parameter ADD specifies whether FUNC will be called additionally to the other bound function or whether it will replace the previous function. See bind for the return value.

bindtags (*tagList=None*)
Set or get the list of bindtags for this widget.

With no argument return the list of all bindtags associated with this widget. With a list of strings as argument the bindtags are set to this list. The bindtags determine in which order events are processed (see bind).

cget (*key*)
Return the resource value for a KEY given as string.

client (*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

clipboard_append(*string*, ***kw*)
Append STRING to the Tk clipboard.

A widget specified at the optional displayof keyword argument specifies the target display. The clipboard can be retrieved with selection_get.

clipboard_clear(***kw*)
Clear the data in the Tk clipboard.

A widget specified for the optional displayof keyword argument specifies the target display.

clipboard_get(***kw*)
Retrieve data from the clipboard on window's display.

The window keyword defaults to the root window of the Tkinter application.

The type keyword specifies the form in which the data is to be returned and should be an atom name such as STRING or FILE_NAME. Type defaults to STRING, except on X11, where the default is to try UTF8_STRING and fall back to STRING.

This command is equivalent to:

selection_get(CLIPBOARD)

colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

colormodel(*value=None*)
Useless. Not implemented in Tk.

columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

config(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

configure(*cnf=None*, ***kw*)
Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

deletecommand(*name*)
Internal function.

Delete the Tcl command provided in NAME.

destroy()
Destroy this and all descendants widgets.

event_add(*virtual*, **sequences*)
Bind a virtual event VIRTUAL (of the form <<Name>>) to an event SEQUENCE such that the virtual event is triggered whenever SEQUENCE occurs.

event_delete(*virtual*, **sequences*)
Unbind a virtual event VIRTUAL from SEQUENCE.

event_generate(*sequence*, ***kw*)
Generate an event SEQUENCE. Additional keyword arguments specify parameter of the event (e.g. x, y, rootx, rooty).

event_info(*virtual=None*)
Return a list of all virtual events or the information about the SEQUENCE bound to the virtual event VIRTUAL.

focus()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focus_displayof()
Return the widget which has currently the focus on the display where this widget is located.

Return None if the application does not have the focus.

focus_force()
Direct input focus to this widget even if the application does not have the focus. Use with caution!

focus_get()
Return the widget which has currently the focus in the application.

Use focus_displayof to allow working with several displays. Return None if application does not have the focus.

focus_lastfor()
Return the widget which would have the focus if top level for this widget gets the focus from the window manager.

focus_set()
Direct input focus to this widget.

If the application currently does not have the focus this widget will get the focus if the application gets the focus through the window manager.

focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

frame()
Return identifier for decorative frame of this widget if present.

geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

getboolean(*s*)
Return a boolean value for Tcl boolean values true and false given as parameter.

getdouble
alias of `__builtin__.float`

getint
alias of `__builtin__.int`

getvar(*name='PY_VAR'*)
Return value of Tcl variable NAME.

grab_current()
Return widget which has currently the grab in this application or None.

grab_release()
Release grab for this widget if currently set.

grab_set(*timeout=30*)

grab_set_global()
Set global grab for this widget.

A global grab directs all events to this and descendant widgets on the display. Use with caution - other applications do not get events anymore.

grab_status()
Return None, "local" or "global" if this widget has no, a local or a global grab.

grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

grid_bbox(*column=None*, *row=None*, *col2=None*, *row2=None*)
Return a tuple of integer coordinates for the bounding box of this widget controlled by the geometry manager grid.

If COLUMN, ROW is given the bounding box applies from the cell with row and column 0 to the specified cell. If COL2 and ROW2 are given the bounding box starts at that cell.

The returned integers specify the offset of the upper left corner in the master widget and the width and height.

grid_columnconfigure(*index*, *cnf={}*, ***kw*)
Configure column INDEX of a grid.

Valid resources are minsize (minimum size of the column), weight (how much does additional space propagate to this column) and pad (how much space to let additionally).

grid_location(*x*, *y*)
Return a tuple of column and row which identify the cell at which the pixel at position X and Y inside the master widget is located.

grid_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given, the current setting will be returned.

grid_rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

grid_size()
Return a tuple of the number of column and rows in the grid.

grid_slaves(*row=None*, *column=None*)
Return a list of all slaves of this widget in its packing order.

group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

iconify()
Display widget as icon.

iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

image_names()
Return a list of all existing image names.

image_types()
Return a list of all available image types (e.g. photo bitmap).

keys()
Return a list of all resource names of this widget.

lift(*aboveThis=None*)
Raise this widget in the stacking order.

lower(*belowThis=None*)
Lower this widget in the stacking order.

mainloop(*n=0*)
Call the mainloop of Tk.

maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

nametowidget(*name*)
Return the Tkinter instance of a widget identified by its Tcl name NAME.

option_add(*pattern*, *value*, *priority=None*)
Set a VALUE (second parameter) for an option PATTERN (first parameter).

An optional third parameter gives the numeric priority (defaults to 80).

option_clear()
Clear the option database.

It will be reloaded if option_add is called.

option_get(*name*, *className*)
Return the value for an option NAME for this widget with CLASSNAME.

Values with higher priority override lower values.

option_readfile(*fileName*, *priority=None*)
Read file FILENAME into the option database.

An optional second parameter gives the numeric priority.

overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

pack_propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

pack_slaves()
Return a list of all slaves of this widget in its packing order.

place_slaves()
Return a list of all slaves of this widget in its packing order.

positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

propagate(*flag=['_noarg_']*)
Set or get the status for propagation of geometry information.

A boolean argument specifies whether the geometry information of the slaves will determine the size of this widget. If no argument is given the current setting will be returned.

protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. "WM_SAVE_YOURSELF" or "WM_DELETE_WINDOW".

quit()
Quit the Tcl interpreter. All widgets will be destroyed.

register(*func*, *subst=None*, *needcleanup=1*)
Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be executed before FUNC.

resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

rowconfigure(*index*, *cnf={}*, ***kw*)
Configure row INDEX of a grid.

Valid resources are minsize (minimum size of the row), weight (how much does additional space propagate to this row) and pad (how much space to let additionally).

selection_clear(***kw*)
Clear the current X selection.

selection_get(***kw*)
Return the contents of the current X selection.

A keyword parameter selection specifies the name of the selection and defaults to PRIMARY. A keyword parameter displayof specifies a widget on the display to use. A keyword parameter type specifies the form of data to be fetched, defaulting to STRING except on X11, where UTF8_STRING is tried before STRING.

selection_handle(*command*, ***kw*)
Specify a function COMMAND to call if the X selection owned by this widget is queried by another application.

This function must return the contents of the selection. The function will be called with the arguments OFFSET and LENGTH which allows the chunking of very long selections. The following keyword parameters can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

selection_own(***kw*)
Become owner of X selection.

A keyword parameter selection specifies the name of the selection (default PRIMARY).

selection_own_get(***kw*)
Return owner of X selection.

The following keyword parameter can be provided: selection - name of the selection (default PRIMARY), type - type of the selection (e.g. STRING, FILE_NAME).

send(*interp*, *cmd*, **args*)
Send Tcl command CMD to different interpreter INTERP to be executed.

setvar(*name='PY_VAR'*, *value='1'*)
Set Tcl variable NAME to VALUE.

show()

size()
Return a tuple of the number of column and rows in the grid.

sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

slaves()
Return a list of all slaves of this widget in its packing order.

state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

title(*string=None*)
Set the title of this widget.

tk_bisque()
Change the color scheme to light brown as used in Tk 3.6 and before.

tk_focusFollowsMouse()
The widget under mouse will get automatically focus. Can not be disabled easily.

tk_focusNext()
Return the next widget in the focus order which follows widget which has currently the focus.

The focus order first goes to the next child, then to the children of the child recursively and then to the next sibling which is higher in the stacking order. A widget is omitted if it has the takefocus resource set to 0.

tk_focusPrev()
Return previous widget in the focus order. See tk_focusNext for details.

tk_menuBar(**args*)
Do not use. Needed in Tk 3.6 and earlier.

tk_setPalette(**args*, ***kw*)
Set a new color scheme for all widget elements.

A single color as argument will cause that all colors of Tk widget elements are derived from this. Alternatively several keyword parameters and its associated colors can be given. The following keywords are valid: activeBackground, foreground, selectColor, activeForeground, highlightBackground, selectBackground, background, highlightColor, selectForeground, disabledForeground, insertBackground, troughColor.

tk_strictMotif(*boolean=None*)
Set Tcl internal variable, whether the look and feel should adhere to Motif.

A parameter of 1 means adhere to Motif (e.g. no color change if mouse passes over slider). Returns the set value.

tkraise(*aboveThis=None*)
Raise this widget in the stacking order.

transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

unbind(*sequence*, *funcid=None*)
Unbind for this widget for event SEQUENCE the function identified with FUNCID.

unbind_all(*sequence*)
Unbind for all widgets for event SEQUENCE all functions.

unbind_class(*className*, *sequence*)
Unbind for all widgets with bindtag CLASSNAME for event SEQUENCE all functions.

update()
Enter event loop until all pending events have been processed by Tcl.

update_idletasks()
Enter event loop until all idle callbacks have been called. This will update the display of windows but not process events caused by the user.

wait_variable(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

wait_visibility(*window=None*)
Wait until the visibility of a WIDGET changes (e.g. it appears).

If no parameter is given self is used.

wait_window(*window=None*)
Wait until a WIDGET is destroyed.

If no parameter is given self is used.

waitvar(*name='PY_VAR'*)
Wait until the variable is modified.

A parameter of type IntVar, StringVar, DoubleVar or BooleanVar must be given.

winfo_atom(*name*, *displayof=0*)
Return integer which represents atom NAME.

winfo_atomname(*id*, *displayof=0*)
Return name of atom with identifier ID.

winfo_cells()
Return number of cells in the colormap for this widget.

winfo_children()
Return a list of all widgets which are children of this widget.

winfo_class()
Return window class name of this widget.

winfo_colormapfull()
Return true if at the last color request the colormap was full.

winfo_containing(*rootX*, *rootY*, *displayof=0*)
Return the widget which is at the root coordinates ROOTX, ROOTY.

winfo_depth()
Return the number of bits per pixel.

winfo_exists()
Return true if this widget exists.

winfo_fpixels(*number*)
Return the number of pixels for the given distance NUMBER (e.g. “3c”) as float.

winfo_geometry()
Return geometry string for this widget in the form “widthxheight+X+Y”.

winfo_height()
Return height of this widget.

winfo_id()
Return identifier ID for this widget.

winfo_interps(*displayof=0*)
Return the name of all Tcl interpreters for this display.

winfo_ismapped()
Return true if this widget is mapped.

winfo_manager()
Return the window manager name for this widget.

winfo_name()
Return the name of this widget.

winfo_parent()
Return the name of the parent of this widget.

winfo_pathname(*id*, *displayof=0*)
Return the pathname of the widget given by ID.

winfo_pixels(*number*)
Rounded integer value of winfo_fpixels.

winfo_pointerx()
Return the x coordinate of the pointer on the root window.

winfo_pointerxy()
Return a tuple of x and y coordinates of the pointer on the root window.

winfo_pointery()
Return the y coordinate of the pointer on the root window.

winfo_reqheight()
Return requested height of this widget.

winfo_reqwidth()
Return requested width of this widget.

winfo_rgb(*color*)
Return tuple of decimal values for red, green, blue for COLOR in this widget.

winfo_rootx()
Return x coordinate of upper left corner of this widget on the root window.

winfo_rooty()
Return y coordinate of upper left corner of this widget on the root window.

winfo_screen()
Return the screen name of this widget.

winfo_screencells()
Return the number of the cells in the colormap of the screen of this widget.

winfo_screendepth()
Return the number of bits per pixel of the root window of the screen of this widget.

winfo_screenheight()
Return the number of pixels of the height of the screen of this widget in pixel.

winfo_screenmmheight()
Return the number of pixels of the height of the screen of this widget in mm.

winfo_screenmmwidth()
Return the number of pixels of the width of the screen of this widget in mm.

winfo_screenvisual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the default colormodel of this screen.

winfo_screenwidth()
Return the number of pixels of the width of the screen of this widget in pixel.

winfo_server()
Return information of the X-Server of the screen of this widget in the form "XmajorRminor vendor vendorVersion".

winfo_toplevel()
Return the toplevel widget of this widget.

winfo_viewable()
Return true if the widget and all its higher ancestors are mapped.

winfo_visual()
Return one of the strings directcolor, grayscale, pseudocolor, staticcolor, staticgray, or truecolor for the colormodel of this widget.

winfo_visualid()
Return the X identifier for the visual for this widget.

winfo_visualsavailable(*includeids=0*)
Return a list of all visuals available for the screen of this widget.

Each item in the list consists of a visual name (see winfo_visual), a depth and if INCLUDEIDS=1 is given also the X identifier.

winfo_vrootheight()
Return the height of the virtual root window associated with this widget in pixels. If there is no virtual root window return the height of the screen.

winfo_vrootwidth()
Return the width of the virtual root window associated with this widget in pixel. If there is no virtual root window return the width of the screen.

winfo_vrootx()
Return the x offset of the virtual root relative to the root window of the screen of this widget.

winfo_vrooty()
Return the y offset of the virtual root relative to the root window of the screen of this widget.

winfo_width()
Return the width of this widget.

winfo_x()
Return the x coordinate of the upper left corner of this widget in the parent.

winfo_y()
Return the y coordinate of the upper left corner of this widget in the parent.

withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

wm_aspect(*minNumer=None*, *minDenom=None*, *maxNumer=None*, *maxDenom=None*)
Instruct the window manager to set the aspect ratio (width/height) of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple of the actual values if no argument is given.

wm_attributes(**args*)
This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and their values. The second form returns the value for the specific option. The third form sets one or more of the values. The values are as follows:

On Windows, -disabled gets or sets whether the window is in a disabled state. -toolwindow gets or sets the style of the window to toolwindow (as defined in the MSDN). -topmost gets or sets whether this is a topmost window (displays above all other windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

wm_client(*name=None*)
Store NAME in WM_CLIENT_MACHINE property of this widget. Return current value.

wm_colormapwindows(**wlist*)
Store list of window names (WLIST) into WM_COLORMAPWINDOWS property of this widget. This list contains windows whose colormaps differ from their parents. Return current list of widgets if WLIST is empty.

wm_command(*value=None*)
Store VALUE in WM_COMMAND property. It is the command which shall be used to invoke the application. Return current command if VALUE is None.

wm_deiconify()
Deiconify this widget. If it was never mapped it will not be mapped. On Windows it will raise this widget and give it the focus.

wm_focusmodel(*model=None*)
Set focus model to MODEL. "active" means that this widget will claim the focus itself, "passive" means that the window manager shall give the focus. Return current focus model if MODEL is None.

wm_frame()
Return identifier for decorative frame of this widget if present.

wm_geometry(*newGeometry=None*)
Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return current value if None is given.

wm_grid(*baseWidth=None*, *baseHeight=None*, *widthInc=None*, *heightInc=None*)
Instruct the window manager that this widget shall only be resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the number of grid units requested in Tk_GeometryRequest.

wm_group(*pathName=None*)
Set the group leader widgets for related widgets to PATHNAME. Return the group leader of this widget if None is given.

wm_iconbitmap(*bitmap=None*, *default=None*)
Set bitmap for the iconified widget to BITMAP. Return the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon for the widget and any descendents that don't have an icon set explicitly. DEFAULT can be the relative path to a .ico file (example: root.iconbitmap(default='myicon.ico')). See Tk documentation for more information.

wm_iconify()
Display widget as icon.

wm_iconmask(*bitmap=None*)
Set mask for the icon bitmap of this widget. Return the mask if None is given.

wm_iconname(*newName=None*)
Set the name of the icon for this widget. Return the name if None is given.

wm_iconposition(*x=None*, *y=None*)
Set the position of the icon of this widget to X and Y. Return a tuple of the current values of X and X if None is given.

wm_iconwindow(*pathName=None*)
Set widget PATHNAME to be displayed instead of icon. Return the current value if None is given.

wm_maxsize(*width=None*, *height=None*)
Set max WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_minsize(*width=None*, *height=None*)
Set min WIDTH and HEIGHT for this widget. If the window is gridded the values are given in grid units. Return the current values if None is given.

wm_overrideredirect(*boolean=None*)
Instruct the window manager to ignore this widget if BOOLEAN is given with 1. Return the current value if None is given.

wm_positionfrom(*who=None*)
Instruct the window manager that the position of this widget shall be defined by the user if WHO is "user", and by its own policy if WHO is "program".

wm_protocol(*name=None*, *func=None*)
Bind function FUNC to command NAME for this widget. Return the function bound to NAME if None is given. NAME could be e.g. “WM_SAVE_YOURSELF” or “WM_DELETE_WINDOW”.

wm_resizable(*width=None*, *height=None*)
Instruct the window manager whether this width can be resized in WIDTH or HEIGHT. Both values are boolean values.

wm_sizefrom(*who=None*)
Instruct the window manager that the size of this widget shall be defined by the user if WHO is “user”, and by its own policy if WHO is “program”.

wm_state(*newstate=None*)
Query or set the state of this widget as one of normal, icon, iconic (see wm_iconwindow), withdrawn, or zoomed (Windows only).

wm_title(*string=None*)
Set the title of this widget.

wm_transient(*master=None*)
Instruct the window manager that this widget is transient with regard to widget MASTER.

wm_withdraw()
Withdraw this widget from the screen such that it is unmapped and forgotten by the window manager. Re-draw it with wm_deiconify.

robot.model package

Package with generic, reusable and extensible model classes.

This package contains, for example, *TestSuite*, *TestCase*, *Keyword* and *SuiteVisitor* base classes. These classes are extended both by *execution* and *result* related model objects and used also elsewhere.

This package is considered stable.

Submodules

robot.model.body module

class robot.model.body.**BodyItem**
Bases: *robot.model.modelobject.ModelObject*

KEYWORD = 'KEYWORD'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

IF = 'IF'

ELSE_IF = 'ELSE IF'

ELSE = 'ELSE'

MESSAGE = 'MESSAGE'

type = None

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters attributes – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters attributes – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

parent

repr_args = ()

class robot.model.body.**Body**(*parent=None*, *items=None*)
Bases: *robot.model.itemlist.ItemList*

A list-like object representing body of a suite, a test or a keyword.

Body contains the keywords and other structures such as for loops.

keyword_class
alias of *robot.model.keyword.Keyword*

for_class
alias of *robot.model.control.For*

if_class
alias of *robot.model.control.If*

classmethod register(*item_class*)

create

create_keyword(**args*, ***kwargs*)

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

append(*item*)

clear()

count(*item*)

extend(*items*)

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.model.body.**IfBranches**(*parent=None*, *items=None*)
Bases: *robot.model.body.Body*

if_branch_class
alias of *robot.model.control.IfBranch*

keyword_class = None

for_class = None

if_class = None

create_branch(**args*, ***kwargs*)

append(*item*)

clear()

count(*item*)

create

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

extend(*items*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF

constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

robot.model.configurer module

class robot.model.configurer.**SuiteConfigurer**(*name=None*, *doc=None*, *metadata=None*, *set_tags=None*, *include_tags=None*, *exclude_tags=None*, *include_suites=None*, *include_tests=None*, *empty_suite_ok=False*)

Bases: *robot.model.visitor.SuiteVisitor*

add_tags

remove_tags

visit_suite(*suite*)

Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

end_for(*for_*)

Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)

Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)

Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)

Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)

Called when keyword ends. Default implementation does nothing.

end_message(*msg*)

Called when message ends. Default implementation does nothing.

end_suite(*suite*)

Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.model.control module

class robot.model.control.**For**(*variables=()*, *flavor='IN'*, *values=()*, *parent=None*)
Bases: *robot.model.body.BodyItem*

type = 'FOR'

body_class
alias of *robot.model.body.Body*

repr_args = ('variables', 'flavor', 'values')

variables

flavor

values

parent

body

keywords
Deprecated since Robot Framework 4.0. Use *body* instead.

visit(*visitor*)

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

class robot.model.control.**If**(*parent=None*)
Bases: *robot.model.body.BodyItem*

IF/ELSE structure root. Branches are stored in *body*.

type = 'IF/ELSE ROOT'

body_class
alias of *robot.model.body.IfBranches*

parent

body

id
Root IF/ELSE id is always `None`.

visit(*visitor*)

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

repr_args = ()

class robot.model.control.**IfBranch**(*type='IF'*, *condition=None*, *parent=None*)
Bases: *robot.model.body.BodyItem*

body_class
alias of *robot.model.body.Body*

repr_args = ('type', 'condition')

type

condition

parent

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

body

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

id
Branch id omits the root IF/ELSE object from the parent id part.

visit(*visitor*)

robot.model.filter module

class robot.model.filter.**EmptySuiteRemover**(*preserve_direct_children=False*)
Bases: *robot.model.visitor.SuiteVisitor*

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

class robot.model.filter.**Filter**(*include_suites=None*, *include_tests=None*, *include_tags=None*, *exclude_tags=None*)
Bases: *robot.model.filter.EmptySuiteRemover*

include_suites

include_tests

include_tags

exclude_tags

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.model.fixture module

robot.model.fixture.**create_fixture**(*fixture*, *parent*, *type*)

robot.model.itemlist module

class robot.model.itemlist.**ItemList**(*item_class*, *common_attrs=None*, *items=None*)
Bases: object

create(**args*, ***kwargs*)

append(*item*)

extend(*items*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

index(*item*, **start_and_end*)

clear()

visit(*visitor*)

count(*item*)

sort()

reverse()

robot.model.keyword module

class robot.model.keyword.**Keyword**(*name=''*, *doc=''*, *args=()*, *assign=()*, *tags=()*, *timeout=None*, *type='KEYWORD'*, *parent=None*)

Bases: *robot.model.body.BodyItem*

Base model for a single keyword.

Extended by *robot.running.model.Keyword* and *robot.result.model.Keyword*.

repr_args = ('name', 'args', 'assign')

doc

args

assign

timeout

type

parent

name

teardown

Keyword teardown as a *Keyword* object.

This attribute is a `Keyword` object also when a keyword has no teardown but in that case its truth value is `False`.

Teardown can be modified by setting attributes directly:

```
keyword.teardown.name = 'Example'
keyword.teardown.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
keyword.teardown.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole teardown is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
keyword.teardown = None
```

New in Robot Framework 4.0. Earlier teardown was accessed like `keyword.keywords.teardown`.

tags

Keyword tags as a *Tags* object.

visit(*visitor*)

Visitor interface entry-point.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

class robot.model.keyword.**Keywords**(*parent=None*, *keywords=None*)
Bases: *robot.model.itemlist.ItemList*

A list-like object representing keywords in a suite, a test or a keyword.

Read-only and deprecated since Robot Framework 4.0.

deprecation_message = "'keywords' attribute is read-only and deprecated since Robot Fra

setup

create_setup(**args*, ***kwargs*)

teardown

create_teardown(**args*, ***kwargs*)

all
Iterates over all keywords, including setup and teardown.

normal
Iterates over normal keywords, omitting setup and teardown.

create(**args*, ***kwargs*)

append(*item*)

extend(*items*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

clear()

count(*item*)

index(*item*, **start_and_end*)

visit(*visitor*)

sort()

reverse()

classmethod raise_deprecation_error()

robot.model.message module

class robot.model.message.**Message**(*message=''*, *level='INFO'*, *html=False*, *timestamp=None*, *parent=None*)

Bases: *robot.model.body.BodyItem*

A message created during the test execution.

Can be a log message triggered by a keyword, or a warning or an error that occurred during parsing or test execution.

type = 'MESSAGE'

repr_args = ('message', 'level')

message
The message content as a string.

level
Severity of the message. Either `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FAIL` or "SKIP". The last two are only used with keyword failure messages.

html
`True` if the content is in HTML, `False` otherwise.

timestamp
Timestamp in format `%Y%m%d %H:%M:%S.%f`.

parent
The object this message was triggered by.

html_message
Returns the message content as HTML.

id

visit(*visitor*)
Visitor interface entry-point.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

class robot.model.message.**Messages**(*message_class=<class 'robot.model.message.Message'>*, *parent=None*, *messages=None*)
Bases: *robot.model.itemlist.ItemList*

append(*item*)

clear()

count(*item*)

create(**args*, ***kwargs*)

extend(*items*)

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

robot.model.metadata module

class robot.model.metadata.**Metadata**(*initial=None*)
Bases: *robot.utils.normalizing.NormalizedDict*

clear() → None. Remove all items from D.

copy()

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

robot.model.modelobject module

class robot.model.modelobject.**ModelObject**
Bases: `object`

repr_args = ()

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

robot.model.modifier module

class robot.model.modifier.**ModelModifier**(*visitors*, *empty_suite_ok*, *logger*)
Bases: *robot.model.visitor.SuiteVisitor*

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.model.namepatterns module

class robot.model.namepatterns.**SuiteNamePatterns**(*patterns=None*)
Bases: robot.model.namepatterns._NamePatterns

match(*name*, *longname=None*)

class robot.model.namepatterns.**TestNamePatterns**(*patterns=None*)
Bases: robot.model.namepatterns._NamePatterns

match(*name*, *longname=None*)

robot.model.statistics module

class robot.model.statistics.**Statistics**(*suite*, *suite_stat_level=-1*, *tag_stat_include=None*, *tag_stat_exclude=None*, *tag_stat_combine=None*, *tag_doc=None*, *tag_stat_link=None*, *rpa=False*)
Bases: object

Container for total, suite and tag statistics.

Accepted parameters have the same semantics as the matching command line options.

total = None
Instance of *TotalStatistics*.

suite = None
Instance of *SuiteStatistics*.

tags = None
Instance of *TagStatistics*.

visit(*visitor*)

class robot.model.statistics.**StatisticsBuilder**(*total_builder*, *suite_builder*, *tag_builder*)
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *`start_for()`* or *`end_for()`* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *`start_for_iteration()`* or *`end_for_iteration()`* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.model.stats module

class robot.model.stats.**Stat**(*name*)
Bases: *robot.utils.sortable.Sortable*

Generic statistic object used for storing all the statistic values.

name = None
Human readable identifier of the object these statistics belong to. *All Tests* for *TotalStatistics*, long name of the suite for *SuiteStatistics* or name of the tag for *TagStatistics*

passed = None
Number of passed tests.

failed = None
Number of failed tests.

skipped = None
Number of skipped tests.

elapsed = None
Number of milliseconds it took to execute.

get_attributes(*include_label=False*, *include_elapsed=False*, *exclude_empty=True*, *values_as_strings=False*, *html_escape=False*)

total

add_test(*test*)

visit(*visitor*)

class robot.model.stats.**TotalStat**(*name*)
Bases: *robot.model.stats.Stat*

Stores statistic values for a test run.

type = 'total'

add_test(*test*)

get_attributes(*include_label=False*, *include_elapsed=False*, *exclude_empty=True*, *values_as_strings=False*, *html_escape=False*)

total

visit(*visitor*)

class robot.model.stats.**SuiteStat**(*suite*)
Bases: *robot.model.stats.Stat*

Stores statistics values for a single suite.

type = 'suite'

id = None
Identifier of the suite, e.g. *s1-s2*.

elapsed = None
Number of milliseconds it took to execute this suite, including sub-suites.

add_stat(*other*)

add_test(*test*)

get_attributes(*include_label=False*, *include_elapsed=False*, *exclude_empty=True*, *values_as_strings=False*, *html_escape=False*)

total

visit(*visitor*)

class robot.model.stats.**TagStat**(*name*, *doc=''*, *links=None*, *combined=None*)
Bases: *robot.model.stats.Stat*

Stores statistic values for a single tag.

type = 'tag'

doc = None
Documentation of tag as a string.

links = None
List of tuples in which the first value is the link URL and the second is the link title. An empty list by default.

combined = None
Pattern as a string if the tag is combined, `None` otherwise.

info
Returns additional information of the tag statistics are about. Either *combined* or an empty string.

add_test(*test*)

get_attributes(*include_label=False*, *include_elapsed=False*, *exclude_empty=True*, *values_as_strings=False*, *html_escape=False*)

total

visit(*visitor*)

class robot.model.stats.**CombinedTagStat**(*pattern*, *name=None*, *doc=''*, *links=None*)
Bases: *robot.model.stats.TagStat*

match(*tags*)

add_test(*test*)

get_attributes(*include_label=False*, *include_elapsed=False*, *exclude_empty=True*, *values_as_strings=False*, *html_escape=False*)

info
Returns additional information of the tag statistics are about. Either *combined* or an empty string.

total

type = 'tag'

visit(*visitor*)

robot.model.suitestatistics module

class robot.model.suitestatistics.**SuiteStatistics**(*suite*)
Bases: object

Container for suite statistics.

stat = None
Instance of *SuiteStat*.

suites = None
List of *TestSuite* objects.

visit(*visitor*)

class robot.model.suitestatistics.**SuiteStatisticsBuilder**(*suite_stat_level*)
Bases: object

current

start_suite(*suite*)

add_test(*test*)

end_suite()

robot.model.tags module

class robot.model.tags.**Tags**(*tags=None*)
Bases: object

add(*tags*)

remove(*tags*)

match(*tags*)

class robot.model.tags.**TagPatterns**(*patterns*)
Bases: object

match(*tags*)

robot.model.tags.**TagPattern**(*pattern*)

class robot.model.tags.**SingleTagPattern**(*pattern*)
Bases: object

match(*tags*)

class robot.model.tags.**AndTagPattern**(*patterns*)
Bases: object

match(*tags*)

class robot.model.tags.**OrTagPattern**(*patterns*)
Bases: object

match(*tags*)

class robot.model.tags.**NotTagPattern**(*must_match*, **must_not_match*)
Bases: object

match(*tags*)

robot.model.tagsetter module

class robot.model.tagsetter.**TagSetter**(*add=None*, *remove=None*)
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*keyword*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.model.tagstatistics module

class robot.model.tagstatistics.**TagStatistics**(*combined_stats*)
Bases: `object`

Container for tag statistics.

tags = None
Dictionary, where key is the name of the tag as a string and value is an instance of *TagStat*.

combined = None
List of *CombinedTagStat* objects.

visit(*visitor*)

class robot.model.tagstatistics.**TagStatisticsBuilder**(*included=None*, *excluded=None*, *combined=None*, *docs=None*, *links=None*)
Bases: `object`

add_test(*test*)

class robot.model.tagstatistics.**TagStatInfo**(*docs=None*, *links=None*)
Bases: `object`

get_stat(*tag*)

get_combined_stats(*combined=None*)

get_doc(*tag*)

get_links(*tag*)

class robot.model.tagstatistics.**TagStatDoc**(*pattern*, *doc*)
Bases: `object`

match(*tag*)

class robot.model.tagstatistics.**TagStatLink**(*pattern*, *link*, *title*)
Bases: `object`

match(*tag*)

get_link(*tag*)

robot.model.testcase module

class robot.model.testcase.**TestCase**(*name=''*, *doc=''*, *tags=None*, *timeout=None*, *parent=None*)

Bases: *robot.model.modelobject.ModelObject*

Base model for a single test case.

Extended by *robot.running.model.TestCase* and *robot.result.model.TestCase*.

body_class
alias of *robot.model.body.Body*

fixture_class
alias of *robot.model.keyword.Keyword*

repr_args = ('name',)

name

doc

timeout

parent

body
Test case body as a *Body* object.

tags
Test tags as a *Tags* object.

setup
Test setup as a *Keyword* object.

This attribute is a `Keyword` object also when a test has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
test.setup.name = 'Example'
test.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
test.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
test.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `test.keywords.setup`.

teardown
Test teardown as a *Keyword* object.

See *setup* for more information.

keywords
Deprecated since Robot Framework 4.0

Use *body*, *setup* or *teardown* instead.

id
Test case id in format like `s1-t3`.

See *TestSuite.id* for more information.

longname
Test name prefixed with the long name of the parent suite.

source

visit(*visitor*)
Visitor interface entry-point.

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

class robot.model.testcase.**TestCases**(*test_class=<class 'robot.model.testcase.TestCase'>*, *parent=None*, *tests=None*)
Bases: *robot.model.itemlist.ItemList*

append(*item*)

clear()

count(*item*)

create(**args*, ***kwargs*)

extend(*items*)

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

robot.model.testsuite module

class robot.model.testsuite.**TestSuite**(*name=''*, *doc=''*, *metadata=None*, *source=None*, *rpa=False*, *parent=None*)

Bases: *robot.model.modelobject.ModelObject*

Base model for single suite.

Extended by *robot.running.model.TestSuite* and *robot.result.model.TestSuite*.

test_class
alias of *robot.model.testcase.TestCase*

fixture_class
alias of *robot.model.keyword.Keyword*

repr_args = ('name',)

doc

source
Path to the source file or directory.

parent
Parent suite. `None` with the root suite.

rpa
`True` when RPA mode is enabled.

name
Test suite name. If not set, constructed from child suite names.

longname
Suite name prefixed with the long name of the parent suite.

metadata
Free test suite metadata as a dictionary.

suites
Child suites as a *TestSuites* object.

tests
Tests as a *TestCases* object.

setup
Suite setup as a *Keyword* object.

This attribute is a `Keyword` object also when a suite has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
suite.setup.name = 'Example'
suite.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
suite.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
suite.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `suite.keywords.setup`.

teardown
Suite teardown as a *Keyword* object.

See *setup* for more information.

keywords
Deprecated since Robot Framework 4.0

Use *setup* or *teardown* instead.

id
An automatically generated unique id.

The root suite has id `s1`, its child suites have ids `s1-s1`, `s1-s2`, ..., their child suites get ids `s1-s1-s1`, `s1-s1-s2`, ..., `s1-s2-s1`, ..., and so on.

The first test in a suite has an id like `s1-t1`, the second has an id `s1-t2`, and so on. Similarly keywords in suites (setup/teardown) and in tests get ids like `s1-k1`, `s1-t1-k1`, and `s1-s4-t2-k5`.

test_count
Number of the tests in this suite, recursively.

has_tests

set_tags(*add=None*, *remove=None*, *persist=False*)
Add and/or remove specified tags to the tests in this suite.

Parameters

- **add** – Tags to add as a list or, if adding only one, as a single string.
- **remove** – Tags to remove as a list or as a single string. Can be given as patterns where `*` and `?` work as wildcards.
- **persist** – Add/remove specified tags also to new tests added to this suite in the future.

filter(*included_suites=None*, *included_tests=None*, *included_tags=None*, *excluded_tags=None*)
Select test cases and remove others from this suite.

Parameters have the same semantics as `--suite`, `--test`, `--include`, and `--exclude` command line options. All of them can be given as a list of strings, or when selecting only one, as a single string.

Child suites that contain no tests after filtering are automatically removed.

Example:

```
suite.filter(included_tests=['Test 1', '* Example'],
             included_tags='priority-1')
```

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

configure(***options*)
A shortcut to configure a suite using one method call.

Can only be used with the root test suite.

Parameters **options** – Passed to *SuiteConfigurer* that will then set suite attributes, call *filter()*, etc. as needed.

Not to be confused with *config()* method that suites, tests, and keywords have to make it possible to set multiple attributes in one call.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

remove_empty_suites(*preserve_direct_children=False*)
Removes all child suites not containing any tests, recursively.

visit(*visitor*)
Visitor interface entry-point.

class robot.model.testsuite.**TestSuites**(*suite_class=<class 'robot.model.testsuite.TestSuite'>*, *parent=None*, *suites=None*)
Bases: *robot.model.itemlist.ItemList*

append(*item*)

clear()

count(*item*)

create(**args*, ***kwargs*)

extend(*items*)

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

robot.model.totalstatistics module

class robot.model.totalstatistics.**TotalStatistics**(*rpa=False*)
Bases: `object`

Container for total statistics.

visit(*visitor*)

total

passed

skipped

failed

add_test(*test*)

message
String representation of the statistics.

For example:: 2 tests, 1 passed, 1 failed

class robot.model.totalstatistics.**TotalStatisticsBuilder**(*suite=None*, *rpa=False*)
Bases: *robot.model.visitor.SuiteVisitor*

add_test(*test*)

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.model.visitor module

Interface to ease traversing through a test suite structure.

Visitors make it easy to modify test suite structures or to collect information from them. They work both with the *executable model* and the *result model*, but the objects passed to the visitor methods are slightly different depending on the model they are used with. The main differences are that on the execution side keywords do not have child keywords nor messages, and that only the result objects have status related attributes like `status` and `starttime`.

This module contains *SuiteVisitor* that implements the core logic to visit a test suite structure, and the *result* package contains *ResultVisitor* that supports visiting the whole test execution result structure. Both of these visitors should be imported via the *robot.api* package when used by external code.

Visitor algorithm

All suite, test, keyword and message objects have a `visit()` method that accepts a visitor instance. These methods will then call the correct visitor method *visit_suite()*, *visit_test()*, *visit_keyword()* or *visit_message()*, depending on the instance where the `visit()` method exists.

The recommended and definitely easiest way to implement a visitor is extending the *SuiteVisitor* base class. The default implementation of its `visit_x()` methods take care of traversing child elements of the object `x` recursively. A `visit_x()` method first calls a corresponding `start_x()` method (e.g. `visit_suite()` calls `start_suite()`), then calls `visit()` for all child objects of the `x` object, and finally calls the corresponding `end_x()` method. The default implementations of `start_x()` and `end_x()` do nothing.

Visitors extending the *SuiteVisitor* can stop visiting at a certain level either by overriding suitable `visit_x()` method or by returning an explicit `False` from any `start_x()` method.

Examples

The following example visitor modifies the test suite structure it visits. It could be used, for example, with Robot Framework's `--prerunmodifier` option to modify test data before execution.

```
"""Pre-run modifier that selects only every Xth test for execution.

Starts from the first test by default. Tests are selected per suite.
"""

from robot.api import SuiteVisitor


class SelectEveryXthTest(SuiteVisitor):

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

(continues on next page)

(continued from previous page)

```
    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
```

For more examples it is possible to look at the source code of visitors used internally by Robot Framework itself. Some good examples are *TagSetter* and *keyword removers*.

class robot.model.visitor.**SuiteVisitor**

Bases: object

Abstract class to ease traversing through the test suite structure.

See the *module level* documentation for more information and an example.

visit_suite(*suite*)

Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

start_suite(*suite*)

Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)

Called when suite ends. Default implementation does nothing.

visit_test(*test*)

Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

start_test(*test*)

Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_test(*test*)

Called when test ends. Default implementation does nothing.

visit_keyword(*kw*)

Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

start_keyword(*keyword*)

Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_keyword(*keyword*)

Called when keyword ends. Default implementation does nothing.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

robot.output package

Package for internal logging and other output.

Not part of the public API, and also subject to change in the future when test execution is refactored.

Subpackages

robot.output.console package

robot.output.console.**ConsoleOutput**(*type='verbose'*, *width=78*, *colors='AUTO'*, *markers='AUTO'*, *stdout=None*, *stderr=None*)

Submodules

robot.output.console.dotted module

class robot.output.console.dotted.**DottedOutput**(*width=78*, *colors='AUTO'*, *stdout=None*, *stderr=None*)
Bases: `object`

start_suite(*suite*)

end_test(*test*)

end_suite(*suite*)

message(*msg*)

output_file(*name*, *path*)

class robot.output.console.dotted.**StatusReporter**(*stream*, *width*)
Bases: *robot.model.visitor.SuiteVisitor*

report(*suite*)

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.output.console.highlighting module

class robot.output.console.highlighting.**HighlightingStream**(*stream*, *colors='AUTO'*)
Bases: `object`

write(*text*, *flush=True*)

flush()

highlight(*text*, *status=None*, *flush=True*)

error(*message*, *level*)

robot.output.console.highlighting.**Highlighter**(*stream*)

class robot.output.console.highlighting.**AnsiHighlighter**(*stream*)
Bases: `object`

green()

red()

yellow()

reset()

class robot.output.console.highlighting.**NoHighlighting**(*stream*)
Bases: *robot.output.console.highlighting.AnsiHighlighter*

green()

red()

reset()

yellow()

class robot.output.console.highlighting.**DosHighlighter**(*stream*)
Bases: object

green()

red()

yellow()

reset()

robot.output.console.quiet module

class robot.output.console.quiet.**QuietOutput**(*colors='AUTO'*, *stderr=None*)
Bases: object

message(*msg*)

class robot.output.console.quiet.**NoOutput**
Bases: object

robot.output.console.verbose module

class robot.output.console.verbose.**VerboseOutput**(*width=78*, *colors='AUTO'*, *markers='AUTO'*, *stdout=None*, *stderr=None*)
Bases: object

start_suite(*suite*)

end_suite(*suite*)

start_test(*test*)

end_test(*test*)

start_keyword(*kw*)

end_keyword(*kw*)

message(*msg*)

output_file(*name*, *path*)

class robot.output.console.verbose.**VerboseWriter**(*width=78*, *colors='AUTO'*, *markers='AUTO'*, *stdout=None*, *stderr=None*)
Bases: object

info(*name*, *doc*, *start_suite=False*)

suite_separator()

test_separator()

status(*status*, *clear=False*)

message(*message*)

keyword_marker(*status*)

error(*message*, *level*, *clear=False*)

output(*name*, *path*)

class robot.output.console.verbose.**KeywordMarker**(*highlighter*, *markers*)
Bases: object

mark(*status*)

reset_count()

Submodules

robot.output.debugfile module

robot.output.debugfile.**DebugFile**(*path*)

robot.output.filelogger module

class robot.output.filelogger.**FileLogger**(*path*, *level*)
Bases: *robot.output.loggerhelper.AbstractLogger*

message(*msg*)

start_suite(*suite*)

end_suite(*suite*)

start_test(*test*)

end_test(*test*)

start_keyword(*kw*)

end_keyword(*kw*)

output_file(*name*, *path*)

close()

debug(*msg*)

error(*msg*)

fail(*msg*)

info(*msg*)

set_level(*level*)

skip(*msg*)

trace(*msg*)

warn(*msg*)

write(*message*, *level*, *html=False*)

robot.output.librarylogger module

Implementation of the public test library logging API.

This is exposed via *robot.api.logger*. Implementation must reside here to avoid cyclic imports.

robot.output.librarylogger.**write**(*msg*, *level*, *html=False*)

robot.output.librarylogger.**trace**(*msg*, *html=False*)

robot.output.librarylogger.**debug**(*msg*, *html=False*)

robot.output.librarylogger.**info**(*msg*, *html=False*, *also_console=False*)

robot.output.librarylogger.**warn**(*msg*, *html=False*)

robot.output.librarylogger.**error**(*msg*, *html=False*)

robot.output.librarylogger.**console**(*msg*, *newline=True*, *stream='stdout'*)

robot.output.listenerarguments module

class robot.output.listenerarguments.**ListenerArguments**(*arguments*)
Bases: object

get_arguments(*version*)

classmethod by_method_name(*name*, *arguments*)

class robot.output.listenerarguments.**MessageArguments**(*arguments*)
Bases: *robot.output.listenerarguments.ListenerArguments*

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**StartSuiteArguments**(*arguments*)
Bases: robot.output.listenerarguments._ListenerArgumentsFromItem

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**EndSuiteArguments**(*arguments*)
Bases: *robot.output.listenerarguments.StartSuiteArguments*

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**StartTestArguments**(*arguments*)
Bases: robot.output.listenerarguments._ListenerArgumentsFromItem

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**EndTestArguments**(*arguments*)
Bases: *robot.output.listenerarguments.StartTestArguments*

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**StartKeywordArguments**(*arguments*)
Bases: robot.output.listenerarguments._ListenerArgumentsFromItem

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

class robot.output.listenerarguments.**EndKeywordArguments**(*arguments*)
Bases: *robot.output.listenerarguments.StartKeywordArguments*

classmethod by_method_name(*name*, *arguments*)

get_arguments(*version*)

robot.output.listenermethods module

class robot.output.listenermethods.**ListenerMethods**(*method_name*, *listeners*)
Bases: object

class robot.output.listenermethods.**LibraryListenerMethods**(*method_name*)
Bases: object

new_suite_scope()

discard_suite_scope()

register(*listeners*, *library*)

unregister(*library*)

class robot.output.listenermethods.**ListenerMethod**(*method*, *listener*, *library=None*)
Bases: object

called = False

robot.output.listeners module

class robot.output.listeners.**Listeners**(*listeners*, *log_level='INFO'*)
Bases: object

set_log_level(*level*)

start_keyword(*kw*)

end_keyword(*kw*)

log_message(*msg*)

imported(*import_type*, *name*, *attrs*)

output_file(*file_type*, *path*)

class robot.output.listeners.**LibraryListeners**(*log_level='INFO'*)
Bases: object

register(*listeners*, *library*)

unregister(*library*, *close=False*)

new_suite_scope()

discard_suite_scope()

set_log_level(*level*)

log_message(*msg*)

imported(*import_type*, *name*, *attrs*)

output_file(*file_type*, *path*)

class robot.output.listeners.**ListenerProxy**(*listener*, *method_names*, *prefix=None*)
Bases: *robot.output.loggerhelper.AbstractLoggerProxy*

classmethod import_listeners(*listeners*, *method_names*, *prefix=None*, *raise_on_error=False*)

robot.output.logger module

class robot.output.logger.**Logger**(*register_console_logger=True*)
Bases: *robot.output.loggerhelper.AbstractLogger*

A global logger proxy to delegating messages to registered loggers.

Whenever something is written to LOGGER in code, all registered loggers are notified. Messages are also cached and cached messages written to new loggers when they are registered.

NOTE: This API is likely to change in future versions.

start_loggers

end_loggers

register_console_logger(*type='verbose'*, *width=78*, *colors='AUTO'*, *markers='AUTO'*, *stdout=None*, *stderr=None*)

unregister_console_logger()

register_syslog(*path=None*, *level='INFO'*)

register_xml_logger(*logger*)

unregister_xml_logger()

register_listeners(*listeners*, *library_listeners*)

register_logger(**loggers*)

unregister_logger(**loggers*)

disable_message_cache()

register_error_listener(*listener*)

message(*msg*)
Messages about what the framework is doing, warnings, errors, ...

cache_only

delayed_logging

log_message(*msg*)
Messages about what the framework is doing, warnings, errors, ...

log_output(*output*)

enable_library_import_logging()

disable_library_import_logging()

start_suite(*suite*)

end_suite(*suite*)

start_test(*test*)

end_test(*test*)

start_keyword(*keyword*)

end_keyword(*keyword*)

imported(*import_type*, *name*, ***attrs*)

output_file(*file_type*, *path*)
Finished output, report, log, debug, or xunit file

close()

debug(*msg*)

error(*msg*)

fail(*msg*)

info(*msg*)

set_level(*level*)

skip(*msg*)

trace(*msg*)

warn(*msg*)

write(*message*, *level*, *html=False*)

class robot.output.logger.**LoggerProxy**(*logger*, *method_names=None*, *prefix=None*)
Bases: *robot.output.loggerhelper.AbstractLoggerProxy*

start_keyword(*kw*)

end_keyword(*kw*)

robot.output.loggerhelper module

class robot.output.loggerhelper.**AbstractLogger**(*level='TRACE'*)
Bases: object

set_level(*level*)

trace(*msg*)

debug(*msg*)

info(*msg*)

warn(*msg*)

fail(*msg*)

skip(*msg*)

error(*msg*)

write(*message*, *level*, *html=False*)

message(*msg*)

class robot.output.loggerhelper.**Message**(*message*, *level='INFO'*, *html=False*, *timestamp=None*)

Bases: *robot.model.message.Message*

message

resolve_delayed_message()

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)

Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)

Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)

Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

html

html_message

Returns the message content as HTML.

id

level

parent

repr_args = ('message', 'level')

timestamp

type = 'MESSAGE'

visit(*visitor*)
Visitor interface entry-point.

class robot.output.loggerhelper.**IsLogged**(*level*)
Bases: object

set_level(*level*)

class robot.output.loggerhelper.**AbstractLoggerProxy**(*logger*, *method_names=None*, *prefix=None*)
Bases: object

robot.output.output module

class robot.output.output.**Output**(*settings*)
Bases: robot.output.loggerhelper.AbstractLogger

register_error_listener(*listener*)

close(*result*)

start_suite(*suite*)

end_suite(*suite*)

start_test(*test*)

end_test(*test*)

start_keyword(*kw*)

end_keyword(*kw*)

message(*msg*)

set_log_level(*level*)

debug(*msg*)

error(*msg*)

fail(*msg*)

info(*msg*)

set_level(*level*)

skip(*msg*)

trace(*msg*)

warn(*msg*)

write(*message*, *level*, *html=False*)

robot.output.pyloggingconf module

robot.output.pyloggingconf.**robot_handler_enabled**(**args*, ***kwds*)

robot.output.pyloggingconf.**set_level**(*level*)

class robot.output.pyloggingconf.**RobotHandler**(*level=0*)
Bases: logging.Handler

Initializes the instance - basically setting the formatter to None and the filter list to empty.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

acquire()
Acquire the I/O thread lock.

addFilter(*filter*)
Add the specified filter to this handler.

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()
Acquire a thread lock for serializing access to the underlying I/O.

filter(*record*)
Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

flush()
Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)
Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)
Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

name

release()
Release the I/O thread lock.

removeFilter(*filter*)
Remove the specified filter from this handler.

setFormatter(*fmt*)
Set the formatter for this handler.

setLevel(*level*)
Set the logging level of this handler.

set_name(*name*)

robot.output.stdoutlogsplitter module

class robot.output.stdoutlogsplitter.**StdoutLogSplitter**(*output*)
Bases: object

Splits messages logged through stdout (or stderr) into Message objects

robot.output.xmllogger module

class robot.output.xmllogger.**XmlLogger**(*path*, *log_level='TRACE'*, *rpa=False*, *generator='Robot'*)
Bases: *robot.result.visitor.ResultVisitor*

close()

set_log_level(*level*)

message(*msg*)

log_message(*msg*)

start_keyword(*kw*)
Called when keyword starts. Default implementation does nothing.

Can return explicit False to stop visiting.

end_keyword(*kw*)
Called when keyword ends. Default implementation does nothing.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit False to stop visiting.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit False to stop visiting.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit False to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

start_statistics(*stats*)

end_statistics(*stats*)

start_total_statistics(*total_stats*)

end_total_statistics(*total_stats*)

start_tag_statistics(*tag_stats*)

end_tag_statistics(*tag_stats*)

start_suite_statistics(*tag_stats*)

end_suite_statistics(*tag_stats*)

visit_stat(*stat*)

start_errors(*errors=None*)

end_errors(*errors=None*)

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_result(*result*)

end_stat(*stat*)

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_result(*result*)

start_stat(*stat*)

visit_errors(*errors*)

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_result(*result*)

visit_statistics(*stats*)

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_suite_statistics(*stats*)

visit_tag_statistics(*stats*)

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_total_statistics(*stats*)

robot.parsing package

Module implementing test data parsing.

Public API is exposed via the *robot.api.parsing* module. See its documentation for more information and examples. If external code needs to import something from this module directly, issue should be submitted about exposing it explicitly via *robot.api.parsing*.

Subpackages

robot.parsing.lexer package

Submodules

robot.parsing.lexer.blocklexers module

class robot.parsing.lexer.blocklexers.**BlockLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.Lexer*

accepts_more(*statement*)

input(*statement*)

lexer_for(*statement*)

lexer_classes()

lex()

handles(*statement*)

class robot.parsing.lexer.blocklexers.**FileLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.BlockLexer*

lex()

lexer_classes()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**SectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.BlockLexer*

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

lexer_classes()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**SettingSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**VariableSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**TestCaseSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**KeywordSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SettingSectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**CommentSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**ImplicitCommentSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**ErrorSectionLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.SectionLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**TestOrKeywordLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.BlockLexer*

name_type = NotImplemented

accepts_more(*statement*)

input(*statement*)

lexer_classes()

handles(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**TestCaseLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.TestOrKeywordLexer*

name_type = 'TESTCASE NAME'

lex()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lexer_classes()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**KeywordLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.TestOrKeywordLexer*

name_type = 'KEYWORD NAME'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

lexer_classes()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**NestedBlockLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.BlockLexer*

accepts_more(*statement*)

input(*statement*)

handles(*statement*)

lex()

lexer_classes()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**ForLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.NestedBlockLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

class robot.parsing.lexer.blocklexers.**IfLexer**(*ctx*)
Bases: *robot.parsing.lexer.blocklexers.NestedBlockLexer*

handles(*statement*)

lexer_classes()

accepts_more(*statement*)

input(*statement*)

lex()

lexer_for(*statement*)

robot.parsing.lexer.context module

class robot.parsing.lexer.context.**LexingContext**(*settings=None*)
Bases: object

settings_class = None

lex_setting(*statement*)

class robot.parsing.lexer.context.**FileContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.LexingContext*

sections_class = None

setting_section(*statement*)

variable_section(*statement*)

test_case_section(*statement*)

keyword_section(*statement*)

comment_section(*statement*)

keyword_context()

lex_invalid_section(*statement*)

lex_setting(*statement*)

settings_class = None

class robot.parsing.lexer.context.**TestCaseFileContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.FileContext*

sections_class
alias of *robot.parsing.lexer.sections.TestCaseFileSections*

settings_class
alias of *robot.parsing.lexer.settings.TestCaseFileSettings*

test_case_context()

comment_section(*statement*)

keyword_context()

keyword_section(*statement*)

lex_invalid_section(*statement*)

lex_setting(*statement*)

setting_section(*statement*)

test_case_section(*statement*)

variable_section(*statement*)

class robot.parsing.lexer.context.**ResourceFileContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.FileContext*

sections_class
alias of *robot.parsing.lexer.sections.ResourceFileSections*

settings_class
alias of *robot.parsing.lexer.settings.ResourceFileSettings*

comment_section(*statement*)

keyword_context()

keyword_section(*statement*)

lex_invalid_section(*statement*)

lex_setting(*statement*)

setting_section(*statement*)

test_case_section(*statement*)

variable_section(*statement*)

class robot.parsing.lexer.context.**InitFileContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.FileContext*

sections_class
alias of *robot.parsing.lexer.sections.InitFileSections*

settings_class
alias of *robot.parsing.lexer.settings.InitFileSettings*

comment_section(*statement*)

keyword_context()

keyword_section(*statement*)

lex_invalid_section(*statement*)

lex_setting(*statement*)

setting_section(*statement*)

test_case_section(*statement*)

variable_section(*statement*)

class robot.parsing.lexer.context.**TestCaseContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.LexingContext*

template_set

lex_setting(*statement*)

settings_class = None

class robot.parsing.lexer.context.**KeywordContext**(*settings=None*)
Bases: *robot.parsing.lexer.context.LexingContext*

template_set

lex_setting(*statement*)

settings_class = None

robot.parsing.lexer.lexer module

robot.parsing.lexer.lexer.**get_tokens**(*source*, *data_only=False*, *tokenize_variables=False*)
Parses the given source to tokens.

Parameters

- **source** – The source where to read the data. Can be a path to a source file as a string or as `pathlib.Path` object, an already opened file object, or Unicode text containing the date directly. Source files must be UTF-8 encoded.
- **data_only** – When `False` (default), returns all tokens. When set to `True`, omits separators, comments, continuation markers, and other non-data tokens.
- **tokenize_variables** – When `True`, possible variables in keyword arguments and elsewhere are tokenized. See the *tokenize_variables()* method for details.

Returns a generator that yields *Token* instances.

robot.parsing.lexer.lexer.**get_resource_tokens**(*source*, *data_only=False*, *tokenize_variables=False*)
Parses the given source to resource file tokens.

Otherwise same as *get_tokens()* but the source is considered to be a resource file. This affects, for example, what settings are valid.

robot.parsing.lexer.lexer.**get_init_tokens**(*source*, *data_only=False*, *tokenize_variables=False*)
Parses the given source to init file tokens.

Otherwise same as *get_tokens()* but the source is considered to be a suite initialization file. This affects, for example, what settings are valid.

class robot.parsing.lexer.lexer.**Lexer**(*ctx*, *data_only=False*, *tokenize_variables=False*)
Bases: `object`

input(*source*)

get_tokens()

robot.parsing.lexer.sections module

class robot.parsing.lexer.sections.**Sections**
Bases: `object`

setting_markers = ('Settings', 'Setting')

variable_markers = ('Variables', 'Variable')

test_case_markers = ('Test Cases', 'Test Case', 'Tasks', 'Task')

keyword_markers = ('Keywords', 'Keyword')

comment_markers = ('Comments', 'Comment')

setting(*statement*)

variable(*statement*)

test_case(*statement*)

keyword(*statement*)

comment(*statement*)

lex_invalid(*statement*)

class robot.parsing.lexer.sections.**TestCaseFileSections**
Bases: *robot.parsing.lexer.sections.Sections*

test_case(*statement*)

comment(*statement*)

comment_markers = ('Comments', 'Comment')

keyword(*statement*)

keyword_markers = ('Keywords', 'Keyword')

lex_invalid(*statement*)

setting(*statement*)

setting_markers = ('Settings', 'Setting')

test_case_markers = ('Test Cases', 'Test Case', 'Tasks', 'Task')

variable(*statement*)

variable_markers = ('Variables', 'Variable')

class robot.parsing.lexer.sections.**ResourceFileSections**
Bases: *robot.parsing.lexer.sections.Sections*

comment(*statement*)

comment_markers = ('Comments', 'Comment')

keyword(*statement*)

keyword_markers = ('Keywords', 'Keyword')

lex_invalid(*statement*)

setting(*statement*)

setting_markers = ('Settings', 'Setting')

test_case(*statement*)

test_case_markers = ('Test Cases', 'Test Case', 'Tasks', 'Task')

variable(*statement*)

variable_markers = ('Variables', 'Variable')

class robot.parsing.lexer.sections.**InitFileSections**
Bases: *robot.parsing.lexer.sections.Sections*

comment(*statement*)

comment_markers = ('Comments', 'Comment')

keyword(*statement*)

keyword_markers = ('Keywords', 'Keyword')

lex_invalid(*statement*)

setting(*statement*)

setting_markers = ('Settings', 'Setting')

test_case(*statement*)

test_case_markers = ('Test Cases', 'Test Case', 'Tasks', 'Task')

variable(*statement*)

variable_markers = ('Variables', 'Variable')

robot.parsing.lexer.settings module

class robot.parsing.lexer.settings.**Settings**
Bases: object

names = ()

aliases = {}

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

lex(*statement*)

class robot.parsing.lexer.settings.**TestCaseFileSettings**
Bases: *robot.parsing.lexer.settings.Settings*

names = ('Documentation', 'Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', '

aliases = {'Task Setup': 'Test Setup', 'Task Teardown': 'Test Teardown', 'Task Templa

lex(*statement*)

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

class robot.parsing.lexer.settings.**InitFileSettings**
Bases: *robot.parsing.lexer.settings.Settings*

names = ('Documentation', 'Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', '

aliases = {}

lex(*statement*)

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

class robot.parsing.lexer.settings.**ResourceFileSettings**
Bases: *robot.parsing.lexer.settings.Settings*

names = ('Documentation', 'Library', 'Resource', 'Variables')

aliases = {}

lex(*statement*)

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

class robot.parsing.lexer.settings.**TestCaseSettings**(*parent*)
Bases: *robot.parsing.lexer.settings.Settings*

names = ('Documentation', 'Tags', 'Setup', 'Teardown', 'Template', 'Timeout')

template_set

aliases = {}

lex(*statement*)

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

class robot.parsing.lexer.settings.**KeywordSettings**
Bases: *robot.parsing.lexer.settings.Settings*

names = ('Documentation', 'Arguments', 'Teardown', 'Timeout', 'Tags', 'Return')

aliases = {}

lex(*statement*)

multi_use = ('Metadata', 'Library', 'Resource', 'Variables')

name_and_arguments = ('Metadata', 'Suite Setup', 'Suite Teardown', 'Test Setup', 'Test

name_arguments_and_with_name = ('Library',)

single_value = ('Resource', 'Test Timeout', 'Test Template', 'Timeout', 'Template')

robot.parsing.lexer.statementlexers module

class robot.parsing.lexer.statementlexers.**Lexer**(*ctx*)
Bases: object

Base class for lexers.

handles(*statement*)

accepts_more(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**StatementLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.Lexer*

token_type = None

accepts_more(*statement*)

input(*statement*)

lex()

handles(*statement*)

class robot.parsing.lexer.statementlexers.**SectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

handles(*statement*)

accepts_more(*statement*)

input(*statement*)

lex()

token_type = None

class robot.parsing.lexer.statementlexers.**SettingSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

token_type = 'SETTING HEADER'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**VariableSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

token_type = 'VARIABLE HEADER'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**TestCaseSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

token_type = 'TESTCASE HEADER'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**KeywordSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

token_type = 'KEYWORD HEADER'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**CommentSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

token_type = 'COMMENT HEADER'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**ErrorSectionHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SectionHeaderLexer*

lex()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**CommentLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

token_type = 'COMMENT'

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

lex()

class robot.parsing.lexer.statementlexers.**SettingLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

lex()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**TestOrKeywordSettingLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.SettingLexer*

handles(*statement*)

accepts_more(*statement*)

input(*statement*)

lex()

token_type = None

class robot.parsing.lexer.statementlexers.**VariableLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

lex()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**KeywordCallLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

lex()

accepts_more(*statement*)

handles(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**ForHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

separators = ('IN', 'IN RANGE', 'IN ENUMERATE', 'IN ZIP')

handles(*statement*)

lex()

accepts_more(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**IfHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

handles(*statement*)

lex()

accepts_more(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**ElseIfHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

handles(*statement*)

lex()

accepts_more(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**ElseHeaderLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

handles(*statement*)

lex()

accepts_more(*statement*)

input(*statement*)

token_type = None

class robot.parsing.lexer.statementlexers.**EndLexer**(*ctx*)
Bases: *robot.parsing.lexer.statementlexers.StatementLexer*

handles(*statement*)

lex()

accepts_more(*statement*)

input(*statement*)

token_type = None

robot.parsing.lexer.tokenizer module

class robot.parsing.lexer.tokenizer.**Tokenizer**
Bases: object

tokenize(*data*, *data_only=False*)

robot.parsing.lexer.tokens module

class robot.parsing.lexer.tokens.**Token**(*type=None*, *value=None*, *lineno=-1*, *col_offset=-1*, *error=None*)
Bases: object

Token representing piece of Robot Framework data.

Each token has type, value, line number, column offset and end column offset in *type*, *value*, *lineno*, *col_offset* and *end_col_offset* attributes, respectively. Tokens representing error also have their error message in *error* attribute.

Token types are declared as class attributes such as *SETTING_HEADER* and *EOL*. Values of these constants have changed slightly in Robot Framework 4.0 and they may change again in the future. It is thus safer to use the constants, not their values, when types are needed. For example, use `Token(Token.EOL)` instead of `Token('EOL')` and `token.type == Token.EOL` instead of `token.type == 'EOL'`.

If *value* is not given when *Token* is initialized and *type* is *IF*, *ELSE_IF*, *ELSE*, *FOR*, *END*, *WITH_NAME* or *CONTINUATION*, the value is automatically set to the correct marker value like `'IF'` or `'ELSE IF'`. If *type* is *EOL* in this case, the value is set to `'\n'`.

SETTING_HEADER = 'SETTING HEADER'

VARIABLE_HEADER = 'VARIABLE HEADER'

TESTCASE_HEADER = 'TESTCASE HEADER'

KEYWORD_HEADER = 'KEYWORD HEADER'

COMMENT_HEADER = 'COMMENT HEADER'

TESTCASE_NAME = 'TESTCASE NAME'

KEYWORD_NAME = 'KEYWORD NAME'

DOCUMENTATION = 'DOCUMENTATION'

```
    SUITE_SETUP = 'SUITE SETUP'
    SUITE_TEARDOWN = 'SUITE TEARDOWN'
    METADATA = 'METADATA'
    TEST_SETUP = 'TEST SETUP'
    TEST_TEARDOWN = 'TEST TEARDOWN'
    TEST_TEMPLATE = 'TEST TEMPLATE'
    TEST_TIMEOUT = 'TEST TIMEOUT'
    FORCE_TAGS = 'FORCE TAGS'
    DEFAULT_TAGS = 'DEFAULT TAGS'
    LIBRARY = 'LIBRARY'
    RESOURCE = 'RESOURCE'
    VARIABLES = 'VARIABLES'
    SETUP = 'SETUP'
    TEARDOWN = 'TEARDOWN'
    TEMPLATE = 'TEMPLATE'
    TIMEOUT = 'TIMEOUT'
    TAGS = 'TAGS'
    ARGUMENTS = 'ARGUMENTS'
    RETURN = 'RETURN'
    NAME = 'NAME'
    VARIABLE = 'VARIABLE'
    ARGUMENT = 'ARGUMENT'
    ASSIGN = 'ASSIGN'
    KEYWORD = 'KEYWORD'
    WITH_NAME = 'WITH NAME'
    FOR = 'FOR'
    FOR_SEPARATOR = 'FOR SEPARATOR'
    END = 'END'
    IF = 'IF'
    ELSE_IF = 'ELSE IF'
    ELSE = 'ELSE'
    SEPARATOR = 'SEPARATOR'
    COMMENT = 'COMMENT'
    CONTINUATION = 'CONTINUATION'
    EOL = 'EOL'
    EOS = 'EOS'
```

ERROR = 'ERROR'

FATAL_ERROR = 'FATAL ERROR'

NON_DATA_TOKENS = frozenset(['COMMENT', 'CONTINUATION', 'SEPARATOR', 'EOS', 'EOL'])

SETTING_TOKENS = frozenset(['RESOURCE', 'TEMPLATE', 'SETUP', 'TAGS', 'SUITE SETUP', 'T

HEADER_TOKENS = frozenset(['VARIABLE HEADER', 'TESTCASE HEADER', 'KEYWORD HEADER', 'CO

ALLOW_VARIABLES = frozenset(['KEYWORD NAME', 'TESTCASE NAME', 'NAME', 'ARGUMENT'])

type

value

lineno

col_offset

error

end_col_offset

set_error(*error*, *fatal=False*)

tokenize_variables()
Tokenizes possible variables in token value.

Yields the token itself if the token does not allow variables (see *Token.ALLOW_VARIABLES*) or its value does not contain variables. Otherwise yields variable tokens as well as tokens before, after, or between variables so that they have the same type as the original token.

class robot.parsing.lexer.tokens.**EOS**(*lineno=-1*, *col_offset=-1*)
Bases: *robot.parsing.lexer.tokens.Token*

Token representing end of a statement.

classmethod from_token(*token*)

ALLOW_VARIABLES = frozenset(['KEYWORD NAME', 'TESTCASE NAME', 'NAME', 'ARGUMENT'])

ARGUMENT = 'ARGUMENT'

ARGUMENTS = 'ARGUMENTS'

ASSIGN = 'ASSIGN'

COMMENT = 'COMMENT'

COMMENT_HEADER = 'COMMENT HEADER'

CONTINUATION = 'CONTINUATION'

DEFAULT_TAGS = 'DEFAULT TAGS'

DOCUMENTATION = 'DOCUMENTATION'

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

END = 'END'

EOL = 'EOL'

EOS = 'EOS'

ERROR = 'ERROR'

```
FATAL_ERROR = 'FATAL ERROR'
FOR = 'FOR'
FORCE_TAGS = 'FORCE TAGS'
FOR_SEPARATOR = 'FOR SEPARATOR'
HEADER_TOKENS = frozenset(['VARIABLE HEADER', 'TESTCASE HEADER', 'KEYWORD HEADER', 'CO
IF = 'IF'
KEYWORD = 'KEYWORD'
KEYWORD_HEADER = 'KEYWORD HEADER'
KEYWORD_NAME = 'KEYWORD NAME'
LIBRARY = 'LIBRARY'
METADATA = 'METADATA'
NAME = 'NAME'
NON_DATA_TOKENS = frozenset(['COMMENT', 'CONTINUATION', 'SEPARATOR', 'EOS', 'EOL'])
RESOURCE = 'RESOURCE'
RETURN = 'RETURN'
SEPARATOR = 'SEPARATOR'
SETTING_HEADER = 'SETTING HEADER'
SETTING_TOKENS = frozenset(['RESOURCE', 'TEMPLATE', 'SETUP', 'TAGS', 'SUITE SETUP', 'T
SETUP = 'SETUP'
SUITE_SETUP = 'SUITE SETUP'
SUITE_TEARDOWN = 'SUITE TEARDOWN'
TAGS = 'TAGS'
TEARDOWN = 'TEARDOWN'
TEMPLATE = 'TEMPLATE'
TESTCASE_HEADER = 'TESTCASE HEADER'
TESTCASE_NAME = 'TESTCASE NAME'
TEST_SETUP = 'TEST SETUP'
TEST_TEARDOWN = 'TEST TEARDOWN'
TEST_TEMPLATE = 'TEST TEMPLATE'
TEST_TIMEOUT = 'TEST TIMEOUT'
TIMEOUT = 'TIMEOUT'
VARIABLE = 'VARIABLE'
VARIABLES = 'VARIABLES'
VARIABLE_HEADER = 'VARIABLE HEADER'
WITH_NAME = 'WITH NAME'
col_offset
```

end_col_offset

error

lineno

set_error(*error*, *fatal=False*)

tokenize_variables()
Tokenizes possible variables in token value.

Yields the token itself if the token does not allow variables (see *Token.ALLOW_VARIABLES*) or its value does not contain variables. Otherwise yields variable tokens as well as tokens before, after, or between variables so that they have the same type as the original token.

type

value

robot.parsing.model package

Submodules

robot.parsing.model.blocks module

class robot.parsing.model.blocks.**Block**
Bases: _ast.AST

errors = ()

lineno

col_offset

end_lineno

end_col_offset

validate_model()

validate()

class robot.parsing.model.blocks.**File**(*sections=None*, *source=None*)
Bases: *robot.parsing.model.blocks.Block*

save(*output=None*)
Save model to the given `output` or to the original source file.

The `output` can be a path to a file or an already opened file object. If `output` is not given, the original source file will be overwritten.

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**Section**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Block*

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**SettingSection**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Section*

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**VariableSection**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Section*

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**TestCaseSection**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Section*

tasks

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**KeywordSection**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Section*

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**CommentSection**(*header=None*, *body=None*)
Bases: *robot.parsing.model.blocks.Section*

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**TestCase**(*header*, *body=None*)
Bases: *robot.parsing.model.blocks.Block*

name

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**Keyword**(*header*, *body=None*)
Bases: *robot.parsing.model.blocks.Block*

name

col_offset

end_col_offset

end_lineno

errors = ()

lineno

validate()

validate_model()

class robot.parsing.model.blocks.**If**(*header*, *body=None*, *orelse=None*, *end=None*, *errors=()*)
Bases: *robot.parsing.model.blocks.Block*

Represents IF structures in the model.

Used with IF, ELSE_IF and ELSE nodes. The *type* attribute specifies the type.

errors = ()

type

condition

validate()

col_offset

end_col_offset

end_lineno

lineno

validate_model()

class robot.parsing.model.blocks.**For**(*header*, *body=None*, *end=None*, *errors=()*)
Bases: *robot.parsing.model.blocks.Block*

errors = ()

variables

values

flavor

validate()

col_offset

end_col_offset

end_lineno

lineno

validate_model()

class robot.parsing.model.blocks.**ModelWriter**(*output*)
Bases: *robot.parsing.model.visitor.ModelVisitor*

write(*model*)

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.parsing.model.blocks.**ModelValidator**
Bases: *robot.parsing.model.visitor.ModelVisitor*

visit_Block(*node*)

visit_Statement(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.parsing.model.blocks.**FirstStatementFinder**
Bases: *robot.parsing.model.visitor.ModelVisitor*

classmethod find_from(*model*)

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.parsing.model.blocks.**LastStatementFinder**
Bases: *robot.parsing.model.visitor.ModelVisitor*

classmethod find_from(*model*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

visit_Statement(*statement*)

robot.parsing.model.statements module

class robot.parsing.model.statements.**Statement**(*tokens*, *errors=()*)
Bases: _ast.AST

type = None

handles_types = ()

lineno

col_offset

end_lineno

end_col_offset

classmethod register(*subcls*)

classmethod from_tokens(*tokens*)

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

data_tokens

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

lines

validate()

class robot.parsing.model.statements.**DocumentationOrMetadata**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

type = None

validate()

class robot.parsing.model.statements.**SingleValue**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

value

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

type = None

validate()

class robot.parsing.model.statements.**MultiValue**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

values

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

type = None

validate()

class robot.parsing.model.statements.**Fixture**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

name

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

type = None

validate()

class robot.parsing.model.statements.**SectionHeader**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

handles_types = ('SETTING HEADER', 'VARIABLE HEADER', 'TESTCASE HEADER', 'KEYWORD HEAD

classmethod from_params(*type*, *name=None*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

type

name

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**LibraryImport**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'LIBRARY'

classmethod from_params(*name*, *args=()*, *alias=None*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

args

alias

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**ResourceImport**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'RESOURCE'

classmethod from_params(*name*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**VariablesImport**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'VARIABLES'

classmethod from_params(*name*, *args=()*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Documentation**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.DocumentationOrMetadata*

type = 'DOCUMENTATION'

classmethod from_params(*value*, *indent=' '*, *separator=' '*, *eol='\n'*, *settings_section=True*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

value

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Metadata**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.DocumentationOrMetadata*

type = 'METADATA'

classmethod from_params(*name*, *value*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

value

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**ForceTags**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.MultiValue*

type = 'FORCE TAGS'

classmethod from_params(*values*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

values

class robot.parsing.model.statements.**DefaultTags**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.MultiValue*

type = 'DEFAULT TAGS'

classmethod from_params(*values*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

values

class robot.parsing.model.statements.**SuiteSetup**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'SUITE SETUP'

classmethod from_params(*name*, *args=()*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**SuiteTeardown**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'SUITE TEARDOWN'

classmethod from_params(*name*, *args=()*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**TestSetup**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'TEST SETUP'

classmethod from_params(*name*, *args=()*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**TestTeardown**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'TEST TEARDOWN'

classmethod from_params(*name*, *args=()*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**TestTemplate**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.SingleValue*

type = 'TEST TEMPLATE'

classmethod from_params(*value*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

value

class robot.parsing.model.statements.**TestTimeout**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.SingleValue*

type = 'TEST TIMEOUT'

classmethod from_params(*value*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

value

class robot.parsing.model.statements.**Variable**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'VARIABLE'

classmethod from_params(*name*, *value*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

value

validate()

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

class robot.parsing.model.statements.**TestCaseName**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'TESTCASE NAME'

classmethod from_params(*name*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**KeywordName**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'KEYWORD NAME'

classmethod from_params(*name*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

name

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Setup**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'SETUP'

classmethod from_params(*name*, *args=()*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Teardown**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Fixture*

type = 'TEARDOWN'

classmethod from_params(*name*, *args=()*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

name

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Tags**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.MultiValue*

type = 'TAGS'

classmethod from_params(*values*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

values

class robot.parsing.model.statements.**Template**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.SingleValue*

type = 'TEMPLATE'

classmethod from_params(*value*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

value

class robot.parsing.model.statements.**Timeout**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.SingleValue*

type = 'TIMEOUT'

classmethod from_params(*value*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

value

class robot.parsing.model.statements.**Arguments**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.MultiValue*

type = 'ARGUMENTS'

classmethod from_params(*args*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

values

class robot.parsing.model.statements.**Return**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.MultiValue*

type = 'RETURN'

classmethod from_params(*args*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

values

class robot.parsing.model.statements.**KeywordCall**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'KEYWORD'

handles_types = ('KEYWORD', 'ASSIGN')

classmethod from_params(*name*, *assign=()*, *args=()*, *indent='   '*, *separator='   '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

keyword

args

assign

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**TemplateArguments**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'ARGUMENT'

classmethod from_params(*args*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

args

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**ForHeader**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'FOR'

classmethod from_params(*variables*, *values*, *flavor='IN'*, *indent='   '*, *separator='   '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

variables

values

flavor

validate()

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

class robot.parsing.model.statements.**IfHeader**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'IF'

classmethod from_params(*condition*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

condition

validate()

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

class robot.parsing.model.statements.**ElseIfHeader**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.IfHeader*

type = 'ELSE IF'

classmethod from_params(*condition*, *indent=' '*, *separator=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

condition

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**ElseHeader**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'ELSE'

classmethod from_params(*indent=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

condition

validate()

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

class robot.parsing.model.statements.**End**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'END'

classmethod from_params(*indent=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

validate()

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

class robot.parsing.model.statements.**Comment**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'COMMENT'

classmethod from_params(*comment*, *indent=' '*, *eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**Error**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'ERROR'

handles_types = ('ERROR', 'FATAL ERROR')

errors
Errors got from the underlying `ERROR` and `FATAL_ERROR` tokens.

Errors can be set also explicitly. When accessing errors, they are returned along with errors got from tokens.

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(**args*, ***kwargs*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

lineno

lines

classmethod register(*subcls*)

validate()

class robot.parsing.model.statements.**EmptyLine**(*tokens*, *errors=()*)
Bases: *robot.parsing.model.statements.Statement*

type = 'EOL'

col_offset

data_tokens

end_col_offset

end_lineno

classmethod from_params(*eol='\n'*)
Create statement from passed parameters.

Required and optional arguments should match class properties. Values are used to create matching tokens.

There is one notable difference for *Documentation* statement where `settings_header` flag is used to determine if statement belongs to settings header or test/keyword.

Most implementations support following general properties: - *separator* whitespace inserted between each token. Default is four spaces. - `indent` whitespace inserted before first token. Default is four spaces. - `eol` end of line sign. Default is `'\n'`.

classmethod from_tokens(*tokens*)

get_token(**types*)
Return a token with the given `type`.

If there are no matches, return `None`. If there are multiple matches, return the first match.

get_tokens(**types*)
Return tokens having any of the given `types`.

get_value(*type*, *default=None*)
Return value of a token with the given `type`.

If there are no matches, return `default`. If there are multiple matches, return the value of the first match.

get_values(**types*)
Return values of tokens having any of the given `types`.

handles_types = ()

lineno

lines

classmethod register(*subcls*)

validate()

robot.parsing.model.visitor module

class robot.parsing.model.visitor.**VisitorFinder**
Bases: `object`

class robot.parsing.model.visitor.**ModelVisitor**
Bases: `ast.NodeVisitor`, *robot.parsing.model.visitor.VisitorFinder*

NodeVisitor that supports matching nodes based on their base classes.

Otherwise identical to the standard ast.NodeVisitor, but allows creating `visit_ClassName` methods so that the `ClassName` is one of the base classes of the node. For example, this visitor method matches all statements:

```
def visit_Statement(self, node):
    # ...
```

visit(*node*)
Visit a node.

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

class robot.parsing.model.visitor.**ModelTransformer**
Bases: ast.NodeTransformer, *robot.parsing.model.visitor.VisitorFinder*

NodeTransformer that supports matching nodes based on their base classes.

See *ModelVisitor* for explanation how this is different compared to the standard ast.NodeTransformer.

visit(*node*)
Visit a node.

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

robot.parsing.parser package

Submodules

robot.parsing.parser.blockparsers module

class robot.parsing.parser.blockparsers.**Parser**(*model*)
Bases: object

Base class for parsers.

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.blockparsers.**BlockParser**(*model*)
Bases: *robot.parsing.parser.blockparsers.Parser*

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.blockparsers.**TestCaseParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.BlockParser*

handles(*statement*)

parse(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

class robot.parsing.parser.blockparsers.**KeywordParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.BlockParser*

handles(*statement*)

parse(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

class robot.parsing.parser.blockparsers.**NestedBlockParser**(*model*)
Bases: *robot.parsing.parser.blockparsers.BlockParser*

handles(*statement*)

parse(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

class robot.parsing.parser.blockparsers.**ForParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.NestedBlockParser*

handles(*statement*)

parse(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

class robot.parsing.parser.blockparsers.**IfParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.NestedBlockParser*

parse(*statement*)

handles(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

class robot.parsing.parser.blockparsers.**OrElseParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.IfParser*

handles(*statement*)

parse(*statement*)

unhandled_tokens = frozenset(['TESTCASE HEADER', 'TESTCASE NAME', 'SETTING HEADER', 'V

robot.parsing.parser.fileparser module

class robot.parsing.parser.fileparser.**FileParser**(*source=None*)
Bases: *robot.parsing.parser.blockparsers.Parser*

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**SectionParser**(*header*)
Bases: *robot.parsing.parser.blockparsers.Parser*

model_class = None

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**SettingSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class
alias of *robot.parsing.model.blocks.SettingSection*

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**VariableSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class
alias of *robot.parsing.model.blocks.VariableSection*

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**CommentSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class
alias of *robot.parsing.model.blocks.CommentSection*

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**ImplicitCommentSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class(*statement*)

handles(*statement*)

parse(*statement*)

class robot.parsing.parser.fileparser.**TestCaseSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class
alias of *robot.parsing.model.blocks.TestCaseSection*

parse(*statement*)

handles(*statement*)

class robot.parsing.parser.fileparser.**KeywordSectionParser**(*header*)
Bases: *robot.parsing.parser.fileparser.SectionParser*

model_class
alias of *robot.parsing.model.blocks.KeywordSection*

parse(*statement*)

handles(*statement*)

robot.parsing.parser.parser module

robot.parsing.parser.parser.**get_model**(*source*, *data_only=False*, *curdir=None*)
Parses the given source to a model represented as an AST.

How to use the model is explained more thoroughly in the general documentation of the *robot.parsing* module.

Parameters

- **source** – The source where to read the data. Can be a path to a source file as a string or as `pathlib.Path` object, an already opened file object, or Unicode text containing the date directly. Source files must be UTF-8 encoded.
- **data_only** – When `False` (default), returns all tokens. When set to `True`, omits separators, comments, continuation markers, and other non-data tokens. Model like this cannot be saved back to file system.
- **curdir** – Directory where the source file exists. This path is used to set the value of the built-in `${CURDIR}` variable during parsing. When not given, the variable is left as-is. Should only be given only if the model will be executed afterwards. If the model is saved back to disk, resolving `${CURDIR}` is typically not a good idea.

Use *get_resource_model()* or *get_init_model()* when parsing resource or suite initialization files, respectively.

robot.parsing.parser.parser.**get_resource_model**(*source*, *data_only=False*, *curdir=None*)
Parses the given source to a resource file model.

Otherwise same as *get_model()* but the source is considered to be a resource file. This affects, for example, what settings are valid.

robot.parsing.parser.parser.**get_init_model**(*source*, *data_only=False*, *curdir=None*)
Parses the given source to a init file model.

Otherwise same as *get_model()* but the source is considered to be a suite initialization file. This affects, for example, what settings are valid.

Submodules

robot.parsing.suitestructure module

class robot.parsing.suitestructure.**SuiteStructure**(*source=None*, *init_file=None*, *children=None*)

Bases: object

is_directory

visit(*visitor*)

class robot.parsing.suitestructure.**SuiteStructureBuilder**(*included_extensions=('robot',)*, *included_suites=None*)

Bases: object

ignored_prefixes = ('_', '.')

ignored_dirs = ('CVS',)

build(*paths*)

class robot.parsing.suitestructure.**SuiteStructureVisitor**

Bases: object

visit_file(*structure*)

visit_directory(*structure*)

start_directory(*structure*)

end_directory(*structure*)

robot.reporting package

Implements report, log, output XML, and xUnit file generation.

The public API of this package is the *ResultWriter* class. It can write result files based on XML output files on the file system, as well as based on the result objects returned by the *ExecutionResult()* factory method or an executed *TestSuite*.

It is highly recommended to use the public API via the *robot.api* package.

This package is considered stable.

Submodules

robot.reporting.expandkeywordmatcher module

class robot.reporting.expandkeywordmatcher.**ExpandKeywordMatcher**(*expand_keywords*)

Bases: object

match(*kw*)

robot.reporting.jsbuildingcontext module

class robot.reporting.jsbuildingcontext.**JsBuildingContext**(*log_path=None*, *split_log=False*, *expand_keywords=None*, *prune_input=False*)

Bases: object

string(*string*, *escape=True*, *attr=False*)

html(*string*)

relative_source(*source*)

timestamp(*time*)

message_level(*level*)

create_link_target(*msg*)

check_expansion(*kw*)

expand_keywords

link(*msg*)

strings

start_splitting_if_needed(*split=False*)

end_splitting(*model*)

prune_input(***kwds*)

robot.reporting.jsexecutionresult module

class robot.reporting.jsexecutionresult.**JsExecutionResult**(*suite*, *statistics*, *errors*, *strings*, *basemillis=None*, *split_results=None*, *min_level=None*, *expand_keywords=None*)

Bases: object

remove_data_not_needed_in_report()

robot.reporting.jsmodelbuilders module

class robot.reporting.jsmodelbuilders.**JsModelBuilder**(*log_path=None*, *split_log=False*, *expand_keywords=None*, *prune_input_to_save_memory=False*)

Bases: object

build_from(*result_from_xml*)

class robot.reporting.jsmodelbuilders.**SuiteBuilder**(*context*)

Bases: robot.reporting.jsmodelbuilders._Builder

build(*suite*)

class robot.reporting.jsmodelbuilders.**TestBuilder**(*context*)

Bases: robot.reporting.jsmodelbuilders._Builder

build(*test*)

class robot.reporting.jsmodelbuilders.**KeywordBuilder**(*context*)

Bases: robot.reporting.jsmodelbuilders._Builder

build(*item*, *split=False*)

build_keyword(*kw*, *split=False*)

class robot.reporting.jsmodelbuilders.**MessageBuilder**(*context*)

Bases: robot.reporting.jsmodelbuilders._Builder

build(*msg*)

class robot.reporting.jsmodelbuilders.**StatisticsBuilder**

Bases: object

build(*statistics*)

class robot.reporting.jsmodelbuilders.**ErrorsBuilder**(*context*)

Bases: robot.reporting.jsmodelbuilders._Builder

build(*errors*)

class robot.reporting.jsmodelbuilders.**ErrorMessageBuilder**(*context*)

Bases: *robot.reporting.jsmodelbuilders.MessageBuilder*

build(*msg*)

robot.reporting.jswriter module

class robot.reporting.jswriter.**JsResultWriter**(*output*, *start_block='<script type="text/javascript">\n'*, *end_block='</script>\n'*, *split_threshold=9500*)

Bases: object

write(*result*, *settings*)

class robot.reporting.jswriter.**SuiteWriter**(*write_json*, *split_threshold*)

Bases: object

write(*suite*, *variable*)

class robot.reporting.jswriter.**SplitLogWriter**(*output*)
Bases: object

write(*keywords*, *strings*, *index*, *notify*)

robot.reporting.logreportwriters module

class robot.reporting.logreportwriters.**LogWriter**(*js_model*)
Bases: robot.reporting.logreportwriters._LogReportWriter

usage = 'log'

write(*path*, *config*)

class robot.reporting.logreportwriters.**ReportWriter**(*js_model*)
Bases: robot.reporting.logreportwriters._LogReportWriter

usage = 'report'

write(*path*, *config*)

class robot.reporting.logreportwriters.**RobotModelWriter**(*output*, *model*, *config*)
Bases: *robot.htmldata.htmlfilewriter.ModelWriter*

write(*line*)

handles(*line*)

robot.reporting.outputwriter module

class robot.reporting.outputwriter.**OutputWriter**(*output*, *rpa=False*)
Bases: *robot.output.xmllogger.XmlLogger*

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

close()

end_result(*result*)

end_errors(*errors=None*)

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*kw*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_stat(*stat*)

end_statistics(*stats*)

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_suite_statistics(*tag_stats*)

end_tag_statistics(*tag_stats*)

end_test(*test*)
Called when test ends. Default implementation does nothing.

end_total_statistics(*total_stats*)

log_message(*msg*)

message(*msg*)

set_log_level(*level*)

start_errors(*errors=None*)

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*kw*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_result(*result*)

start_stat(*stat*)

start_statistics(*stats*)

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite_statistics(*tag_stats*)

start_tag_statistics(*tag_stats*)

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_total_statistics(*total_stats*)

visit_errors(*errors*)

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_result(*result*)

visit_stat(*stat*)

visit_statistics(*stats*)

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_suite_statistics(*stats*)

visit_tag_statistics(*stats*)

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_total_statistics(*stats*)

robot.reporting.resultwriter module

class robot.reporting.resultwriter.**ResultWriter**(**sources*)
Bases: object

A class to create log, report, output XML and xUnit files.

Parameters **sources** – Either one *Result* object, or one or more paths to existing output XML files.

By default writes report.html and log.html, but no output XML or xUnit files. Custom file names can be given and results disabled or enabled using settings or options passed to the *write_results()* method. The latter is typically more convenient:

```
writer = ResultWriter(result)
writer.write_results(report='custom.html', log=None, xunit='xunit.xml')
```

write_results(*settings=None*, ***options*)
Writes results based on the given settings or options.

Parameters

- **settings** – *RebotSettings* object to configure result writing.
- **options** – Used to construct new *RebotSettings* object if settings are not given.

class robot.reporting.resultwriter.**Results**(*settings*, **sources*)
Bases: object

result

js_result

robot.reporting.stringcache module

class robot.reporting.stringcache.**StringIndex**
Bases: int

bit_length() → int
Number of bits necessary to represent self in binary. >>> bin(37) '0b100101' >>> (37).bit_length() 6

conjugate()
Returns self, the complex conjugate of any int.

denominator
the denominator of a rational number in lowest terms

imag
the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

class robot.reporting.stringcache.**StringCache**
Bases: object

add(*text*)

dump()

robot.reporting.xunitwriter module

class robot.reporting.xunitwriter.**XUnitWriter**(*execution_result*)
Bases: object

write(*output*)

class robot.reporting.xunitwriter.**XUnitFileWriter**(*xml_writer*)
Bases: *robot.result.visitor.ResultVisitor*

Provides an xUnit-compatible result file.

Attempts to adhere to the de facto schema guessed by Peter Reilly, see: http://marc.info/?l=ant-dev&m=123551933508682

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_statistics(*stats*)

visit_errors(*errors*)

end_result(*result*)

end_errors(*errors*)

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_stat(*stat*)

end_statistics(*stats*)

end_suite_statistics(*suite_stats*)

end_tag_statistics(*stats*)

end_test(*test*)
Called when test ends. Default implementation does nothing.

end_total_statistics(*stats*)

start_errors(*errors*)

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_result(*result*)

start_stat(*stat*)

start_statistics(*stats*)

start_suite_statistics(*stats*)

start_tag_statistics(*stats*)

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_total_statistics(*stats*)

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_result(*result*)

visit_stat(*stat*)

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_suite_statistics(*stats*)

visit_tag_statistics(*stats*)

visit_total_statistics(*stats*)

robot.result package

Implements parsing execution results from XML output files.

The main public API of this package consists of the *ExecutionResult()* factory method, that returns *Result* objects, and of the *ResultVisitor* abstract class, that eases further processing the results.

The model objects in the *model* module can also be considered to be part of the public API, because they can be found inside the *Result* object. They can also be inspected and modified as part of the normal test execution by pre-Rebot modifiers and listeners.

It is highly recommended to import the public entry-points via the *robot.api* package like in the example below. In those rare cases where the aforementioned model objects are needed directly, they can be imported from this package.

This package is considered stable.

Example

```
#!/usr/bin/env python

"""Usage: check_test_times.py seconds inpath [outpath]

Reads test execution result from an output XML file and checks that no test
took longer than given amount of seconds to execute.

Optional `outpath` specifies where to write processed results. If not given,
results are written over the original file.
"""

import sys
from robot.api import ExecutionResult, ResultVisitor


class ExecutionTimeChecker(ResultVisitor):

    def __init__(self, max_seconds):
        self.max_milliseconds = max_seconds * 1000

    def visit_test(self, test):
        if test.status == 'PASS' and test.elapsedtime > self.max_milliseconds:
            test.status = 'FAIL'
            test.message = 'Test execution took too long.'


def check_tests(seconds, inpath, outpath=None):
    result = ExecutionResult(inpath)
    result.visit(ExecutionTimeChecker(float(seconds)))
    result.save(outpath)


if __name__ == '__main__':
    try:
        check_tests(*sys.argv[1:])
    except TypeError:
        print(__doc__)
```

Submodules

robot.result.configurer module

class robot.result.configurer.**SuiteConfigurer**(*remove_keywords=None*, *log_level=None*, *start_time=None*, *end_time=None*, ***base_config*)

Bases: *robot.model.configurer.SuiteConfigurer*

Result suite configured.

Calls suite's `remove_keywords()` and `filter_messages()` methods and sets its start and end time based on the given named parameters.

`base_config` is forwarded to *robot.model.SuiteConfigurer* that will do further configuration based on them.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *`start_suite()`* or *`end_suite()`* nor visiting child suites, tests or keywords (setup and teardown) at all.

add_tags

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

remove_tags

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.result.executionerrors module

class robot.result.executionerrors.**ExecutionErrors**(*messages=None*)
Bases: `object`

Represents errors occurred during the execution of tests.

An error might be, for example, that importing a library has failed.

id = 'errors'

messages
A *list-like object* of *Message* instances.

add(*other*)

visit(*visitor*)

robot.result.executionresult module

class robot.result.executionresult.**Result**(*source=None*, *root_suite=None*, *errors=None*, *rpa=None*)

Bases: object

Test execution results.

Can be created based on XML output files using the *ExecutionResult()* factory method. Also returned by the *robot.running.TestSuite.run* method.

source = None
Path to the XML file where results are read from.

suite = None
Hierarchical execution results as a *TestSuite* object.

errors = None
Execution errors as an *ExecutionErrors* object.

statistics
Test execution statistics.

Statistics are an instance of *Statistics* that is created based on the contained suite and possible *configuration*.

Statistics are created every time this property is accessed. Saving them to a variable is thus often a good idea to avoid re-creating them unnecessarily:

```
from robot.api import ExecutionResult

result = ExecutionResult('output.xml')
result.configure(stat_config={'suite_stat_level': 2,
                              'tag_stat_combine': 'tagANDanother'})
stats = result.statistics
print(stats.total.failed)
print(stats.total.passed)
print(stats.tags.combined[0].total)
```

return_code
Return code (integer) of test execution.

By default returns the number of failed tests (max 250), but can be *configured* to always return 0.

configure(*status_rc=True*, *suite_config=None*, *stat_config=None*)
Configures the result object and objects it contains.

Parameters

- **status_rc** – If set to False, *return_code* always returns 0.

- **suite_config** – A dictionary of configuration options passed to `configure()` method of the contained `suite`.
- **stat_config** – A dictionary of configuration options used when creating *statistics*.

save(*path=None*)
Save results as a new output XML file.

Parameters **path** – Path to save results to. If omitted, overwrites the original file.

visit(*visitor*)
An entry point to visit the whole result object.

Parameters **visitor** – An instance of *ResultVisitor*.

Visitors can gather information, modify results, etc. See *result* package for a simple usage example.

Notice that it is also possible to call `result.suite.visit` if there is no need to visit the contained `statistics` or `errors`.

handle_suite_teardown_failures()
Internal usage only.

set_execution_mode(*other*)
Set execution mode based on other result. Internal usage only.

class robot.result.executionresult.**CombinedResult**(*results=None*)
Bases: *robot.result.executionresult.Result*

Combined results of multiple test executions.

add_result(*other*)

configure(*status_rc=True*, *suite_config=None*, *stat_config=None*)
Configures the result object and objects it contains.

Parameters

- **status_rc** – If set to `False`, *return_code* always returns 0.
- **suite_config** – A dictionary of configuration options passed to `configure()` method of the contained `suite`.
- **stat_config** – A dictionary of configuration options used when creating *statistics*.

handle_suite_teardown_failures()
Internal usage only.

return_code
Return code (integer) of test execution.

By default returns the number of failed tests (max 250), but can be *configured* to always return 0.

save(*path=None*)
Save results as a new output XML file.

Parameters **path** – Path to save results to. If omitted, overwrites the original file.

set_execution_mode(*other*)
Set execution mode based on other result. Internal usage only.

statistics
Test execution statistics.

Statistics are an instance of *Statistics* that is created based on the contained `suite` and possible *configuration*.

Statistics are created every time this property is accessed. Saving them to a variable is thus often a good idea to avoid re-creating them unnecessarily:

```
from robot.api import ExecutionResult

result = ExecutionResult('output.xml')
result.configure(stat_config={'suite_stat_level': 2,
                              'tag_stat_combine': 'tagANDanother'})
stats = result.statistics
print(stats.total.failed)
print(stats.total.passed)
print(stats.tags.combined[0].total)
```

visit(*visitor*)

An entry point to visit the whole result object.

Parameters **visitor** – An instance of *ResultVisitor*.

Visitors can gather information, modify results, etc. See *result* package for a simple usage example.

Notice that it is also possible to call `result.suite.visit` if there is no need to visit the contained `statistics` or `errors`.

robot.result.flattenkeywordmatcher module

robot.result.flattenkeywordmatcher.**validate_flatten_keyword**(*options*)

class robot.result.flattenkeywordmatcher.**FlattenByTypeMatcher**(*flatten*)

Bases: `object`

match(*tag*)

class robot.result.flattenkeywordmatcher.**FlattenByNameMatcher**(*flatten*)

Bases: `object`

match(*kwname*, *libname=None*)

class robot.result.flattenkeywordmatcher.**FlattenByTagMatcher**(*flatten*)

Bases: `object`

match(*kwtags*)

robot.result.keywordremover module

robot.result.keywordremover.**KeywordRemover**(*how*)

class robot.result.keywordremover.**AllKeywordsRemover**

Bases: `robot.result.keywordremover._KeywordRemover`

visit_keyword(*keyword*)

Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

class robot.result.keywordremover.**PassedKeywordRemover**
Bases: `robot.result.keywordremover._KeywordRemover`

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*keyword*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *`start_for()`* or *`end_for()`* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *`start_for_iteration()`* or *`end_for_iteration()`* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *`visit_if_branch()`*.

Can be overridden to allow modifying the passed in `if_` without calling *`start_if()`* or *`end_if()`* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *`start_if_branch()`* or *`end_if_branch()`* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *`start_message()`* or *`end_message()`*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *`start_suite()`* or *`end_suite()`* nor visiting child suites, tests or keywords (setup and teardown) at all.

class robot.result.keywordremover.**ByNameKeywordRemover**(*pattern*)
Bases: `robot.result.keywordremover._KeywordRemover`

start_keyword(*kw*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *`start_for()`* or *`end_for()`* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *`start_for_iteration()`* or *`end_for_iteration()`* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *`visit_if_branch()`*.

Can be overridden to allow modifying the passed in `if_` without calling *`start_if()`* or *`end_if()`* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

class robot.result.keywordremover.**ByTagKeywordRemover**(*pattern*)
Bases: robot.result.keywordremover._KeywordRemover

start_keyword(*kw*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

class robot.result.keywordremover.**ForLoopItemsRemover**
Bases: robot.result.keywordremover._KeywordRemover

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *`start_suite()`* or *`end_suite()`* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *`start_test()`* or *`end_test()`* nor visiting keywords.

class robot.result.keywordremover.**WaitUntilKeywordSucceedsRemover**
Bases: `robot.result.keywordremover._KeywordRemover`

start_keyword(*kw*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

class robot.result.keywordremover.**WarningAndErrorFinder**
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

class robot.result.keywordremover.**RemovalMessage**(*message*)
Bases: `object`

set_if_removed(*kw*, *len_before*)

set(*kw*, *message=None*)

robot.result.merger module

class robot.result.merger.**Merger**(*result*, *rpa=False*)
Bases: *robot.model.visitor.SuiteVisitor*

merge(*merged*)

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.result.messagefilter module

class robot.result.messagefilter.**MessageFilter**(*loglevel=None*)
Bases: *robot.model.visitor.SuiteVisitor*

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.result.model module

Module implementing result related model objects.

During test execution these objects are created internally by various runners. At that time they can inspected and modified by listeners.

When results are parsed from XML output files after execution to be able to create logs and reports, these objects are created by the *ExecutionResult()* factory method. At that point they can be inspected and modified by pre-Rebot modifiers.

The *ExecutionResult()* factory method can also be used by custom scripts and tools. In such usage it is often easiest to inspect and modify these objects using the *visitor interface*.

class robot.result.model.**Body**(*parent=None*, *items=None*)
Bases: *robot.model.body.Body*

message_class
alias of *Message*

create_message(**args*, ***kwargs*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *messages=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

append(*item*)

clear()

count(*item*)

create

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

extend(*items*)

for_class
alias of *For*

if_class
alias of *If*

index(*item*, **start_and_end*)

insert(*index*, *item*)

keyword_class
alias of *Keyword*

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.result.model.**ForIterations**(*parent=None*, *items=None*)
Bases: *robot.result.model.Body*

for_iteration_class
alias of *ForIteration*

keyword_class = None

if_class = None

for_class = None

create_iteration(**args*, ***kwargs*)

append(*item*)

clear()

count(*item*)

create

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

create_message(**args*, ***kwargs*)

extend(*items*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *messages=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

index(*item*, **start_and_end*)

insert(*index*, *item*)

message_class
alias of *Message*

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.result.model.**IfBranches**(*parent=None*, *items=None*)
Bases: *robot.result.model.Body*, *robot.model.body.IfBranches*

append(*item*)

clear()

count(*item*)

create

create_branch(**args*, ***kwargs*)

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

create_message(**args*, ***kwargs*)

extend(*items*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *messages=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

for_class
alias of *For*

if_branch_class
alias of *IfBranch*

if_class
alias of *If*

index(*item*, **start_and_end*)

insert(*index*, *item*)

keyword_class
alias of *Keyword*

message_class
alias of *Message*

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.result.model.**Message**(*message=''*, *level='INFO'*, *html=False*, *timestamp=None*, *parent=None*)
Bases: *robot.model.message.Message*

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

html

html_message
Returns the message content as HTML.

id

level

message

parent

repr_args = ('message', 'level')

timestamp

type = 'MESSAGE'

visit(*visitor*)
Visitor interface entry-point.

class robot.result.model.**StatusMixin**
Bases: object

PASS = 'PASS'

FAIL = 'FAIL'

SKIP = 'SKIP'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

elapsedtime
Total execution time in milliseconds.

passed
`True` when `status` is 'PASS', `False` otherwise.

failed
`True` when `status` is 'FAIL', `False` otherwise.

skipped
`True` when `status` is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

not_run
`True` when `status` is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

class robot.result.model.**ForIteration**(*variables=None*, *status='FAIL'*, *starttime=None*, *endtime=None*, *doc=''*, *parent=None*)

Bases: *robot.model.body.BodyItem*, *robot.result.model.StatusMixin*, *robot.result.modeldeprecation.DeprecatedAttributesMixin*

type = 'FOR ITERATION'

body_class
alias of *Body*

repr_args = ('variables',)

variables

parent

status

starttime

endtime

doc

body

visit(*visitor*)

name
Deprecated.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FAIL = 'FAIL'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SETUP = 'SETUP'

SKIP = 'SKIP'

TEARDOWN = 'TEARDOWN'

args
Deprecated.

assign
Deprecated.

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

kwname
Deprecated.

libname
Deprecated.

message
Deprecated.

not_run
`True` when *`status`* is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

passed
`True` when *`status`* is 'PASS', `False` otherwise.

skipped
`True` when *`status`* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

tags
Deprecated.

timeout
Deprecated.

class robot.result.model.**For**(*variables=()*, *flavor='IN'*, *values=()*, *status='FAIL'*, *starttime=None*, *endtime=None*, *doc=''*, *parent=None*)
Bases: *`robot.model.control.For`*, *`robot.result.model.StatusMixin`*, *`robot.result.modeldeprecation.DeprecatedAttributesMixin`*

body_class
alias of *`ForIterations`*

status

starttime

endtime

doc

name
Deprecated.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FAIL = 'FAIL'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SETUP = 'SETUP'

SKIP = 'SKIP'

TEARDOWN = 'TEARDOWN'

args
Deprecated.

assign
Deprecated.

body

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

flavor

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

keywords
Deprecated since Robot Framework 4.0. Use *body* instead.

kwname
Deprecated.

libname
Deprecated.

message
Deprecated.

not_run
`True` when *status* is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

parent

passed
`True` when *status* is 'PASS', `False` otherwise.

repr_args = ('variables', 'flavor', 'values')

skipped
`True` when *status* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

tags
Deprecated.

timeout
Deprecated.

type = 'FOR'

values

variables

visit(*visitor*)

class robot.result.model.**If**(*parent=None*, *status='FAIL'*, *starttime=None*, *endtime=None*, *doc=''*)
Bases: *robot.model.control.If*, *robot.result.model.StatusMixin*, *robot.result.modeldeprecation.DeprecatedAttributesMixin*

body_class
alias of *IfBranches*

status

starttime

endtime

doc

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FAIL = 'FAIL'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SETUP = 'SETUP'

SKIP = 'SKIP'

TEARDOWN = 'TEARDOWN'

args
Deprecated.

assign
Deprecated.

body

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

id
Root IF/ELSE id is always `None`.

kwname
Deprecated.

libname
Deprecated.

message
Deprecated.

name
Deprecated.

not_run
`True` when *status* is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

parent

passed
`True` when *status* is 'PASS', `False` otherwise.

repr_args = ()

skipped
`True` when *status* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

tags
Deprecated.

timeout
Deprecated.

type = 'IF/ELSE ROOT'

visit(*visitor*)

class robot.result.model.**IfBranch**(*type='IF'*, *condition=None*, *status='FAIL'*, *starttime=None*, *endtime=None*, *doc=''*, *parent=None*)
Bases: *robot.model.control.IfBranch*, *robot.result.model.StatusMixin*, *robot.result.modeldeprecation.DeprecatedAttributesMixin*

body_class
alias of *Body*

status

starttime

endtime

doc

name
Deprecated.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FAIL = 'FAIL'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SETUP = 'SETUP'

SKIP = 'SKIP'

TEARDOWN = 'TEARDOWN'

args
Deprecated.

assign
Deprecated.

body

condition

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

id
Branch id omits the root IF/ELSE object from the parent id part.

kwname
Deprecated.

libname
Deprecated.

message
Deprecated.

not_run
`True` when *status* is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

parent

passed
`True` when *status* is 'PASS', `False` otherwise.

repr_args = ('type', 'condition')

skipped
`True` when *status* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

tags
Deprecated.

timeout
Deprecated.

type

visit(*visitor*)

class robot.result.model.**Keyword**(*kwname=''*, *libname=''*, *doc=''*, *args=()*, *assign=()*, *tags=()*, *timeout=None*, *type='KEYWORD'*, *status='FAIL'*, *starttime=None*, *endtime=None*, *parent=None*, *sourcename=None*)

Bases: *robot.model.keyword.Keyword*, *robot.result.model.StatusMixin*

Represents results of a single keyword.

See the base class for documentation of attributes not documented here.

body_class
alias of *Body*

kwname
Name of the keyword without library or resource name.

libname
Name of the library or resource containing this keyword.

status
Execution status as a string. `PASS`, `FAIL`, `SKIP` or `NOT RUN`.

starttime
Keyword execution start time in format `%Y%m%d %H:%M:%S.%f`.

endtime
Keyword execution end time in format `%Y%m%d %H:%M:%S.%f`.

message
Keyword status message. Used only if suite teardowns fails.

sourcename
Original name of keyword with embedded arguments.

body
Child keywords and messages as a *Body* object.

keywords
Deprecated since Robot Framework 4.0.

Use *body* or *teardown* instead.

messages
Keyword's messages.

Starting from Robot Framework 4.0 this is a list generated from messages in *body*.

children
List of child keywords and messages in creation order.

Deprecated since Robot Framework 4.0. Use **:att:'body'** instead.

name
Keyword name in format `libname.kwname`.

Just `kwname` if *libname* is empty. In practice that is the case only with user keywords in the same file as the executed test case or test suite.

Cannot be set directly. Set *libname* and *kwname* separately instead.

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FAIL = 'FAIL'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SETUP = 'SETUP'

SKIP = 'SKIP'

TEARDOWN = 'TEARDOWN'

args

assign

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

not_run
`True` when *status* is 'NOT RUN', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

parent

passed
`True` when *status* is 'PASS', `False` otherwise.

repr_args = ('name', 'args', 'assign')

skipped
`True` when *status* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

tags
Keyword tags as a *Tags* object.

teardown
Keyword teardown as a *Keyword* object.

This attribute is a `Keyword` object also when a keyword has no teardown but in that case its truth value is `False`.

Teardown can be modified by setting attributes directly:

```
keyword.teardown.name = 'Example'
keyword.teardown.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
keyword.teardown.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole teardown is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
keyword.teardown = None
```

New in Robot Framework 4.0. Earlier teardown was accessed like `keyword.keywords.teardown`.

timeout

type

visit(*visitor*)
Visitor interface entry-point.

class robot.result.model.**TestCase**(*name=''*, *doc=''*, *tags=None*, *timeout=None*, *status='FAIL'*, *message=''*, *starttime=None*, *endtime=None*)
Bases: *robot.model.testcase.TestCase*, *robot.result.model.StatusMixin*

Represents results of a single test case.

See the base class for documentation of attributes not documented here.

body_class
alias of *Body*

fixture_class
alias of *Keyword*

status
Status as a string `PASS` or `FAIL`. See also *passed*.

message
Test message. Typically a failure message but can be set also when test passes.

starttime
Test case execution start time in format `%Y%m%d %H:%M:%S.%f`.

endtime
Test case execution end time in format `%Y%m%d %H:%M:%S.%f`.

not_run

critical

FAIL = 'FAIL'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SKIP = 'SKIP'

body
Test case body as a *Body* object.

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

elapsedtime
Total execution time in milliseconds.

failed
`True` when *status* is 'FAIL', `False` otherwise.

id
Test case id in format like `s1-t3`.

See *TestSuite.id* for more information.

keywords
Deprecated since Robot Framework 4.0

Use *body*, *setup* or *teardown* instead.

longname
Test name prefixed with the long name of the parent suite.

name

parent

passed
`True` when *status* is 'PASS', `False` otherwise.

repr_args = ('name',)

setup
Test setup as a *Keyword* object.

This attribute is a `Keyword` object also when a test has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
test.setup.name = 'Example'
test.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
test.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
test.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `test.keywords.setup`.

skipped
`True` when *status* is 'SKIP', `False` otherwise.

Setting to `False` value is ambiguous and raises an exception.

source

tags
Test tags as a *Tags* object.

teardown
Test teardown as a *Keyword* object.

See *setup* for more information.

timeout

visit(*visitor*)
Visitor interface entry-point.

class robot.result.model.**TestSuite**(*name=''*, *doc=''*, *metadata=None*, *source=None*, *message=''*, *starttime=None*, *endtime=None*, *rpa=False*)
Bases: *robot.model.testsuite.TestSuite*, *robot.result.model.StatusMixin*

Represents results of a single test suite.

See the base class for documentation of attributes not documented here.

test_class
alias of *TestCase*

fixture_class
alias of *Keyword*

message
Possible suite setup or teardown error message.

starttime
Suite execution start time in format `%Y%m%d %H:%M:%S.%f`.

endtime
Suite execution end time in format `%Y%m%d %H:%M:%S.%f`.

passed
`True` if no test has failed but some have passed, `False` otherwise.

failed
`True` if any test has failed, `False` otherwise.

skipped
`True` if there are no passed or failed tests, `False` otherwise.

not_run

status
'PASS', 'FAIL' or 'SKIP' depending on test statuses.

- If any test has failed, status is 'FAIL'.
- If no test has failed but at least some test has passed, status is 'PASS'.
- If there are no failed or passed tests, status is 'SKIP'. This covers both the case when all tests have been skipped and when there are no tests.

statistics
Suite statistics as a *TotalStatistics* object.

Recreated every time this property is accessed, so saving the results to a variable and inspecting it is often a good idea:

```
stats = suite.statistics
print(stats.failed)
print(stats.total)
print(stats.message)
```

full_message
Combination of *message* and *stat_message*.

FAIL = 'FAIL'

NOT_RUN = 'NOT RUN'

NOT_SET = 'NOT SET'

PASS = 'PASS'

SKIP = 'SKIP'

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

filter(*included_suites=None*, *included_tests=None*, *included_tags=None*, *excluded_tags=None*)
Select test cases and remove others from this suite.

Parameters have the same semantics as `--suite`, `--test`, `--include`, and `--exclude` command line options. All of them can be given as a list of strings, or when selecting only one, as a single string.

Child suites that contain no tests after filtering are automatically removed.

Example:

```
suite.filter(included_tests=['Test 1', '* Example'],
             included_tags='priority-1')
```

has_tests

id
An automatically generated unique id.

The root suite has id `s1`, its child suites have ids `s1-s1`, `s1-s2`, ..., their child suites get ids `s1-s1-s1`, `s1-s1-s2`, ..., `s1-s2-s1`, ..., and so on.

The first test in a suite has an id like `s1-t1`, the second has an id `s1-t2`, and so on. Similarly keywords in suites (setup/teardown) and in tests get ids like `s1-k1`, `s1-t1-k1`, and `s1-s4-t2-k5`.

keywords
Deprecated since Robot Framework 4.0

Use *setup* or *teardown* instead.

longname
Suite name prefixed with the long name of the parent suite.

metadata
Free test suite metadata as a dictionary.

name
Test suite name. If not set, constructed from child suite names.

parent

remove_empty_suites (*preserve_direct_children=False*)
Removes all child suites not containing any tests, recursively.

repr_args = ('name',)

rpa

set_tags (*add=None*, *remove=None*, *persist=False*)
Add and/or remove specified tags to the tests in this suite.

Parameters

- **add** – Tags to add as a list or, if adding only one, as a single string.
- **remove** – Tags to remove as a list or as a single string. Can be given as patterns where `*` and `?` work as wildcards.
- **persist** – Add/remove specified tags also to new tests added to this suite in the future.

setup
Suite setup as a *Keyword* object.

This attribute is a `Keyword` object also when a suite has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
suite.setup.name = 'Example'
suite.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
suite.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
suite.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `suite.keywords.setup`.

source

stat_message
String representation of the *statistics*.

suites
Child suites as a *TestSuites* object.

teardown
Suite teardown as a *Keyword* object.

See *setup* for more information.

test_count
Number of the tests in this suite, recursively.

tests
Tests as a *TestCases* object.

visit(*visitor*)
Visitor interface entry-point.

elapsedtime
Total execution time in milliseconds.

remove_keywords(*how*)
Remove keywords based on the given condition.

Parameters **how** – What approach to use when removing keywords. Either `ALL`, `PASSED`, `FOR`, `WUKS`, or `NAME:<pattern>`.

For more information about the possible values see the documentation of the `--removekeywords` command line option.

filter_messages(*log_level='TRACE'*)
Remove log messages below the specified `log_level`.

configure(***options*)
A shortcut to configure a suite using one method call.

Can only be used with the root test suite.

Parameters **options** – Passed to *SuiteConfigurer* that will then set suite attributes, call *filter()*, etc. as needed.

Example:

```
suite.configure(remove_keywords='PASSED',
                doc='Smoke test results.')
```

Not to be confused with *config()* method that suites, tests, and keywords have to make it possible to set multiple attributes in one call.

handle_suite_teardown_failures()
Internal usage only.

suite_teardown_failed(*error*)
Internal usage only.

suite_teardown_skipped(*message*)
Internal usage only.

robot.result.modeldeprecation module

robot.result.modeldeprecation.**deprecated**(*method*)

class robot.result.modeldeprecation.**DeprecatedAttributesMixin**
Bases: object

name
Deprecated.

kwname
Deprecated.

libname
Deprecated.

args
Deprecated.

assign
Deprecated.

tags
Deprecated.

timeout
Deprecated.

message
Deprecated.

robot.result.resultbuilder module

robot.result.resultbuilder.**ExecutionResult**(**sources*, ***options*)
Factory method to constructs *Result* objects.

Parameters

- **sources** – XML source(s) containing execution results. Can be specified as paths, opened file objects, or strings/bytes containing XML directly. Support for bytes is new in RF 3.2.
- **options** – Configuration options. Using `merge=True` causes multiple results to be combined so that tests in the latter results replace the ones in the original. Setting `rpa` either to `True` (RPA mode) or `False` (test automation) sets execution mode explicitly. By default it is got from processed output files and conflicting modes cause an error. Other options are passed directly to the *ExecutionResultBuilder* object used internally.

Returns *Result* instance.

Should be imported by external code via the *robot.api* package. See the *robot.result* package for a usage example.

class robot.result.resultbuilder.**ExecutionResultBuilder**(*source*, *include_keywords=True*, *flattened_keywords=None*)
Bases: `object`

Builds *Result* objects based on output files.

Instead of using this builder directly, it is recommended to use the *ExecutionResult()* factory method.

Parameters

- **source** – Path to the XML output file to build *Result* objects from.
- **include_keywords** – Boolean controlling whether to include keyword information in the result or not. Keywords are not needed when generating only report. Although the the option name has word "keyword", it controls also including FOR and IF structures.
- **flatten_keywords** – List of patterns controlling what keywords to flatten. See the documentation of `--flattenkeywords` option for more details.

build(*result*)

class robot.result.resultbuilder.**RemoveKeywords**
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.result.suiteteardownfailed module

class robot.result.suiteteardownfailed.**SuiteTeardownFailureHandler**
Bases: *robot.model.visitor.SuiteVisitor*

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *`start_test()`* or *`end_test()`* nor visiting keywords.

visit_keyword(*keyword*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *`start_keyword()`* or *`end_keyword()`* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

class robot.result.suiteteardownfailed.**SuiteTeardownFailed**(*message*, *skipped=False*)
Bases: *robot.model.visitor.SuiteVisitor*

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*keyword*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.result.visitor module

Visitors can be used to easily traverse result structures.

This module contains *ResultVisitor* for traversing the whole *Result* object. It extends *SuiteVisitor* that contains visiting logic for the test suite structure.

class robot.result.visitor.**ResultVisitor**
Bases: *robot.model.visitor.SuiteVisitor*

Abstract class to conveniently travel *Result* objects.

A visitor implementation can be given to the `visit()` method of a result object. This will cause the result object to be traversed and the visitor's `visit_x()`, `start_x()`, and `end_x()` methods to be called for each suite, test, keyword and message, as well as for errors, statistics, and other information in the result object. See methods below for a full list of available visitor methods.

See the *result package level* documentation for more information about handling results and a concrete visitor example. For more information about the visitor algorithm see documentation in *robot.model.visitor* module.

visit_result(*result*)

start_result(*result*)

end_result(*result*)

visit_statistics(*stats*)

start_statistics(*stats*)

end_statistics(*stats*)

visit_total_statistics(*stats*)

start_total_statistics(*stats*)

end_total_statistics(*stats*)

visit_tag_statistics(*stats*)

start_tag_statistics(*stats*)

end_tag_statistics(*stats*)

visit_suite_statistics(*stats*)

start_suite_statistics(*stats*)

end_suite_statistics(*suite_stats*)

visit_stat(*stat*)

start_stat(*stat*)

end_stat(*stat*)

visit_errors(*errors*)

start_errors(*errors*)

end_errors(*errors*)

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

robot.result.xmlelementhandlers module

class robot.result.xmlelementhandlers.**XmlElementHandler**(*execution_result*, *root_handler=None*)
Bases: `object`

start(*elem*)

end(*elem*)

class robot.result.xmlelementhandlers.**ElementHandler**
Bases: `object`

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

tag = None

children = frozenset([])

classmethod register(*handler*)

get_child_handler(*tag*)

start(*elem*, *result*)

end(*elem*, *result*)

class robot.result.xmlelementhandlers.**RootHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

children = frozenset(['robot'])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

tag = None

class robot.result.xmlelementhandlers.**RobotHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'robot'

children = frozenset(['suite', 'statistics', 'errors'])

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**SuiteHandler**

Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'suite'

children = frozenset(['status', 'doc', 'meta', 'kw', 'test', 'suite', 'metadata'])

start(*elem*, *result*)

get_child_handler(*tag*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**TestHandler**

Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'test'

children = frozenset(['status', 'for', 'tags', 'doc', 'tag', 'kw', 'timeout', 'msg', '

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**KeywordHandler**

Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'kw'

children = frozenset(['status', 'for', 'tags', 'doc', 'msg', 'tag', 'kw', 'arguments',

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**ForHandler**

Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'for'

children = frozenset(['status', 'doc', 'iter', 'msg', 'value', 'var'])

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**ForIterationHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'iter'

children = frozenset(['status', 'for', 'doc', 'msg', 'kw', 'var', 'if'])

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**IfHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'if'

children = frozenset(['status', 'msg', 'branch'])

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**IfBranchHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'branch'

children = frozenset(['status', 'msg', 'kw', 'for', 'if'])

start(*elem*, *result*)

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**MessageHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'msg'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**StatusHandler**(*set_status=True*)
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'status'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**DocHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'doc'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**MetadataHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'metadata'

children = frozenset(['item'])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**MetadataItemHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'item'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**MetaHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'meta'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**TagsHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'tags'

children = frozenset(['tag'])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**TagHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'tag'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**TimeoutHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'timeout'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**AssignHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'assign'

children = frozenset(['var'])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**VarHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'var'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**ArgumentsHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'arguments'

children = frozenset(['arg'])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**ArgumentHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'arg'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**ValueHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'value'

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

class robot.result.xmlelementhandlers.**ErrorsHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'errors'

start(*elem*, *result*)

get_child_handler(*tag*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

classmethod register(*handler*)

class robot.result.xmlelementhandlers.**ErrorMessageHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

end(*elem*, *result*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

get_child_handler(*tag*)

classmethod register(*handler*)

start(*elem*, *result*)

tag = None

class robot.result.xmlelementhandlers.**StatisticsHandler**
Bases: *robot.result.xmlelementhandlers.ElementHandler*

tag = 'statistics'

get_child_handler(*tag*)

children = frozenset([])

element_handlers = {'arg': <robot.result.xmlelementhandlers.ArgumentHandler object>,

end(*elem*, *result*)

classmethod register(*handler*)

start(*elem*, *result*)

robot.running package

Implements the core test execution logic.

The main public entry points of this package are of the following two classes:

- *TestSuiteBuilder* for creating executable test suites based on existing test case files and directories.
- *TestSuite* for creating an executable test suite structure programmatically.

It is recommended to import both of these classes via the *robot.api* package like in the examples below. Also *TestCase* and *Keyword* classes used internally by the *TestSuite* class are part of the public API. In those rare cases where these classes are needed directly, they can be imported from this package.

Examples

First, let's assume we have the following test suite in file `activate_skynet.robot`:

```
*** Settings ***
Library    OperatingSystem

*** Test Cases ***
Should Activate Skynet
    [Tags]    smoke
    [Setup]    Set Environment Variable    SKYNET    activated
    Environment Variable Should Be Set    SKYNET
```

We can easily parse and create an executable test suite based on the above file using the `TestSuiteBuilder` class as follows:

```
from robot.api import TestSuiteBuilder

suite = TestSuiteBuilder().build('path/to/activate_skynet.robot')
```

That was easy. Let's next generate the same test suite from scratch using the *TestSuite* class:

```
from robot.api import TestSuite

suite = TestSuite('Activate Skynet')
suite.resource.imports.library('OperatingSystem')
test = suite.tests.create('Should Activate Skynet', tags=['smoke'])
test.setup.config('Set Environment Variable', args=['SKYNET', 'activated'])
test.keywords.create('Environment Variable Should Be Set', args=['SKYNET'])
```

Not that complicated either, especially considering the flexibility. Notice that the suite created based on the file could also be edited further using the same API.

Now that we have a test suite ready, let's *execute it* and verify that the returned *Result* object contains correct information:

```
result = suite.run(output='skynet.xml')

assert result.return_code == 0
assert result.suite.name == 'Activate Skynet'
test = result.suite.tests[0]
assert test.name == 'Should Activate Skynet'
assert test.passed
```

(continues on next page)

(continued from previous page)

```
stats = result.suite.statistics
assert stats.total == 1 and stats.failed == 0
```

Running the suite generates a normal output XML file, unless it is disabled by using `output=None`. Generating log, report, and xUnit files based on the results is possible using the *`ResultWriter`* class:

```
from robot.api import ResultWriter

# Report and xUnit files can be generated based on the result object.
ResultWriter(result).write_results(report='skynet.html', log=None)
# Generating log files requires processing the earlier generated output XML.
ResultWriter('skynet.xml').write_results()
```

Subpackages

robot.running.arguments package

Submodules

robot.running.arguments.argumentconverter module

class robot.running.arguments.argumentconverter.**ArgumentConverter**(*argspec*, *dry_run=False*)

Bases: object

convert(*positional*, *named*)

robot.running.arguments.argumentmapper module

class robot.running.arguments.argumentmapper.**ArgumentMapper**(*argspec*)

Bases: object

map(*positional*, *named*, *replace_defaults=True*)

class robot.running.arguments.argumentmapper.**KeywordCallTemplate**(*argspec*)

Bases: object

fill_positional(*positional*)

fill_named(*named*)

replace_defaults()

class robot.running.arguments.argumentmapper.**DefaultValue**(*value*)

Bases: object

resolve(*variables*)

robot.running.arguments.argumentparser module

class robot.running.arguments.argumentparser.**JavaArgumentParser**(*type='Keyword'*)
Bases: robot.running.arguments.argumentparser._ArgumentParser

parse(*signatures*, *name=None*)

class robot.running.arguments.argumentparser.**DynamicArgumentParser**(*type='Keyword'*)
Bases: robot.running.arguments.argumentparser._ArgumentSpecParser

parse(*argspec*, *name=None*)

class robot.running.arguments.argumentparser.**UserKeywordArgumentParser**(*type='Keyword'*)
Bases: robot.running.arguments.argumentparser._ArgumentSpecParser

parse(*argspec*, *name=None*)

robot.running.arguments.argumentresolver module

class robot.running.arguments.argumentresolver.**ArgumentResolver**(*argspec*, *resolve_named=True*, *resolve_variables_until=None*, *dict_to_kwargs=False*)
Bases: object

resolve(*arguments*, *variables=None*)

class robot.running.arguments.argumentresolver.**NamedArgumentResolver**(*argspec*)
Bases: object

resolve(*arguments*, *variables=None*)

class robot.running.arguments.argumentresolver.**NullNamedArgumentResolver**
Bases: object

resolve(*arguments*, *variables=None*)

class robot.running.arguments.argumentresolver.**DictToKwargs**(*argspec*, *enabled=False*)
Bases: object

handle(*positional*, *named*)

class robot.running.arguments.argumentresolver.**VariableReplacer**(*resolve_until=None*)
Bases: object

replace(*positional*, *named*, *variables=None*)

robot.running.arguments.argumentspec module

class robot.running.arguments.argumentspec.**Enum**
Bases: object

class robot.running.arguments.argumentspec.**ArgumentSpec**(*name=None*, *type='Keyword'*, *positional_only=None*, *positional_or_named=None*, *var_positional=None*, *named_only=None*, *var_named=None*, *defaults=None*, *types=None*)

Bases: object

types

positional

minargs

maxargs

argument_names

resolve(*arguments*, *variables=None*, *resolve_named=True*, *resolve_variables_until=None*, *dict_to_kwargs=False*)

map(*positional*, *named*, *replace_defaults=True*)

class robot.running.arguments.argumentspec.**ArgInfo**(*kind*, *name=''*, *types=<object object>*, *default=<object object>*)

Bases: object

NOTSET = <object object>

POSITIONAL_ONLY = 'POSITIONAL_ONLY'

POSITIONAL_ONLY_MARKER = 'POSITIONAL_ONLY_MARKER'

POSITIONAL_OR_NAMED = 'POSITIONAL_OR_NAMED'

VAR_POSITIONAL = 'VAR_POSITIONAL'

NAMED_ONLY_MARKER = 'NAMED_ONLY_MARKER'

NAMED_ONLY = 'NAMED_ONLY'

VAR_NAMED = 'VAR_NAMED'

types

required

types_reprs

default_repr

robot.running.arguments.argumentvalidator module

class robot.running.arguments.argumentvalidator.**ArgumentValidator**(*argspec*)

Bases: object

validate(*positional*, *named*, *dryrun=False*)

robot.running.arguments.embedded module

class robot.running.arguments.embedded.**EmbeddedArguments**(*name*)
Bases: object

class robot.running.arguments.embedded.**EmbeddedArgumentParser**
Bases: object

parse(*string*)

robot.running.arguments.javaargumentcoercer module

robot.running.arguments.py2argumentparser module

class robot.running.arguments.py2argumentparser.**PythonArgumentParser**(*type='Keyword'*)
Bases: object

parse(*handler*, *name=None*)

robot.running.arguments.py3argumentparser module

robot.running.arguments.typeconverters module

class robot.running.arguments.typeconverters.**Enum**
Bases: object

class robot.running.arguments.typeconverters.**TypeConverter**(*used_type*)
Bases: object

type = None

type_name = None

abc = None

aliases = ()

value_types = (<type 'unicode'>,)

classmethod register(*converter*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

no_conversion_needed(*value*)

class robot.running.arguments.typeconverters.**StringConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.unicode

type_name = 'string'

aliases = ('string', 'str', 'unicode')

abc = None

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

class robot.running.arguments.typeconverters.**BooleanConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

value_types = (<type 'unicode'>, <type 'int'>, <type 'float'>, <type 'NoneType'>)

type
alias of __builtin__.bool

type_name = 'boolean'

aliases = ('bool',)

abc = None

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**IntegerConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.int

abc
alias of numbers.Integral

type_name = 'integer'

aliases = ('int', 'long')

value_types = (<type 'unicode'>, <type 'float'>)

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**FloatConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.float

abc
alias of numbers.Real

type_name = 'float'

aliases = ('double',)

value_types = (<type 'unicode'>, <class 'numbers.Real'>)

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**DecimalConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of decimal.Decimal

type_name = 'decimal'

value_types = (<type 'unicode'>, <type 'int'>, <type 'float'>)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**BytesConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.str

abc = None

type_name = 'bytes'

value_types = (<type 'unicode'>, <type 'bytearray'>)

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**ByteArrayConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.bytearray

type_name = 'bytearray'

value_types = (<type 'unicode'>, <type 'str'>)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**DateTimeConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of datetime.datetime

type_name = 'datetime'

value_types = (<type 'unicode'>, <type 'int'>, <type 'float'>)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**DateConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of datetime.date

type_name = 'date'

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

class robot.running.arguments.typeconverters.**TimeDeltaConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of datetime.timedelta

type_name = 'timedelta'

value_types = (<type 'unicode'>, <type 'int'>, <type 'float'>)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**EnumConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of *Enum*

type_name

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

class robot.running.arguments.typeconverters.**NoneConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.NoneType

type_name = 'None'

classmethod handles(*type_*)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

class robot.running.arguments.typeconverters.**ListConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.list

type_name = 'list'

abc
alias of _abcoll.Sequence

value_types = (<type 'unicode'>, <type 'tuple'>)

no_conversion_needed(*value*)

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**TupleConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.tuple

type_name = 'tuple'

value_types = (<type 'unicode'>, <type 'list'>)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**DictionaryConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.dict

abc
alias of _abcoll.Mapping

type_name = 'dictionary'

aliases = ('dict', 'map')

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

class robot.running.arguments.typeconverters.**SetConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.set

type_name = 'set'

value_types = (<type 'unicode'>, <type 'frozenset'>, <type 'list'>, <type 'tuple'>, <c

abc
alias of _abcoll.Set

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**FrozenSetConverter**(*used_type*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type
alias of __builtin__.frozenset

type_name = 'frozenset'

value_types = (<type 'unicode'>, <type 'set'>, <type 'list'>, <type 'tuple'>, <class '

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod handles(*type_*)

no_conversion_needed(*value*)

classmethod register(*converter*)

class robot.running.arguments.typeconverters.**CombinedConverter**(*union*)
Bases: *robot.running.arguments.typeconverters.TypeConverter*

type = typing.Union

type_name

classmethod handles(*type_*)

no_conversion_needed(*value*)

abc = None

aliases = ()

convert(*name*, *value*, *explicit_type=True*, *strict=True*)

classmethod converter_for(*type_*)

classmethod register(*converter*)

value_types = (<type 'unicode'>,)

robot.running.arguments.typevalidator module

class robot.running.arguments.typevalidator.**TypeValidator**(*argspec*)
Bases: object

validate(*types*)

validate_type_dict(*types*)

convert_type_list_to_dict(*types*)

robot.running.builder package

Submodules

robot.running.builder.builders module

class robot.running.builder.builders.**TestSuiteBuilder**(*included_suites=None*, *included_extensions=('robot',)*, *rpa=None*, *allow_empty_suite=False*, *process_curdir=True*)
Bases: object

Builder to construct `TestSuite` objects based on data on the disk.

The *build()* method constructs executable *TestSuite* objects based on test data files or directories. There are two main use cases for this API:

- Execute the created suite by using its *run()* method. The suite can be can be modified before execution if needed.
- Inspect the suite to see, for example, what tests it has or what tags tests have. This can be more convenient than using the lower level *parsing* APIs but does not allow saving modified data back to the disk.

Both modifying the suite and inspecting what data it contains are easiest done by using the *visitor* interface.

This class is part of the public API and should be imported via the *robot.api* package.

Parameters

- **include_suites** – List of suite names to include. If `None` or an empty list, all suites are included. Same as using `--suite` on the command line.
- **included_extensions** – List of extensions of files to parse. Same as `--extension`. This parameter was named `extension` before RF 3.2.
- **rpa** – Explicit test execution mode. `True` for RPA and `False` for test automation. By default mode is got from test data headers and possible conflicting headers cause an error. Same as `--rpa` or `--norpa`.

- **allow_empty_suite** – Specify is it an error if the built suite contains no tests. Same as `--runemptysuite`. New in RF 3.2.
- **process_curdir** – Control processing the special `${CURDIR}` variable. It is resolved already at parsing time by default, but that can be changed by giving this argument `False` value. New in RF 3.2.

build(**paths*)

Parameters **paths** – Paths to test data files or directories.

Returns *TestSuite* instance.

class robot.running.builder.builders.**SuiteStructureParser**(*included_extensions*, *rpa=None*, *process_curdir=True*)

Bases: *robot.parsing.suitestructure.SuiteStructureVisitor*

parse(*structure*)

visit_file(*structure*)

start_directory(*structure*)

end_directory(*structure*)

visit_directory(*structure*)

class robot.running.builder.builders.**ResourceFileBuilder**(*process_curdir=True*)

Bases: object

build(*source*)

robot.running.builder.parsers module

class robot.running.builder.parsers.**BaseParser**

Bases: object

parse_init_file(*source*, *defaults=None*)

parse_suite_file(*source*, *defaults=None*)

parse_resource_file(*source*)

class robot.running.builder.parsers.**RobotParser**(*process_curdir=True*)

Bases: *robot.running.builder.parsers.BaseParser*

parse_init_file(*source*, *defaults=None*)

parse_suite_file(*source*, *defaults=None*)

build_suite(*model*, *name=None*, *defaults=None*)

parse_resource_file(*source*)

class robot.running.builder.parsers.**RestParser**(*process_curdir=True*)

Bases: *robot.running.builder.parsers.RobotParser*

build_suite(*model*, *name=None*, *defaults=None*)

parse_init_file(*source*, *defaults=None*)

parse_resource_file(*source*)

parse_suite_file(*source*, *defaults=None*)

class robot.running.builder.parsers.**NoInitFileDirectoryParser**
Bases: *robot.running.builder.parsers.BaseParser*

parse_init_file(*source*, *defaults=None*)

parse_resource_file(*source*)

parse_suite_file(*source*, *defaults=None*)

robot.running.builder.parsers.**format_name**(*source*)

class robot.running.builder.parsers.**ErrorReporter**(*source*)
Bases: ast.NodeVisitor

visit_Error(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

robot.running.builder.testsettings module

class robot.running.builder.testsettings.**TestDefaults**(*parent=None*)
Bases: object

setup

teardown

force_tags

timeout

class robot.running.builder.testsettings.**TestSettings**(*defaults*)
Bases: object

setup

teardown

timeout

template

tags

robot.running.builder.transformers module

class robot.running.builder.transformers.**SettingsBuilder**(*suite*, *test_defaults*)
Bases: ast.NodeVisitor

visit_Documentation(*node*)

visit_Metadata(*node*)

visit_SuiteSetup(*node*)

visit_SuiteTeardown(*node*)

visit_TestSetup(*node*)

visit_TestTeardown(*node*)

visit_TestTimeout(*node*)

visit_DefaultTags(*node*)

visit_ForceTags(*node*)

visit_TestTemplate(*node*)

visit_ResourceImport(*node*)

visit_LibraryImport(*node*)

visit_VariablesImport(*node*)

visit_VariableSection(*node*)

visit_TestCaseSection(*node*)

visit_KeywordSection(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**SuiteBuilder**(*suite*, *test_defaults*)
Bases: ast.NodeVisitor

visit_SettingSection(*node*)

visit_Variable(*node*)

visit_TestCase(*node*)

visit_Keyword(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**ResourceBuilder**(*resource*)
Bases: ast.NodeVisitor

visit_Documentation(*node*)

visit_LibraryImport(*node*)

visit_ResourceImport(*node*)

visit_VariablesImport(*node*)

visit_Variable(*node*)

visit_Keyword(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**TestCaseBuilder**(*suite*, *defaults*)
Bases: ast.NodeVisitor

visit_TestCase(*node*)

visit_For(*node*)

visit_If(*node*)

visit_TemplateArguments(*node*)

visit_Documentation(*node*)

visit_Setup(*node*)

visit_Teardown(*node*)

visit_Timeout(*node*)

visit_Tags(*node*)

visit_Template(*node*)

visit_KeywordCall(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**KeywordBuilder**(*resource*)
Bases: ast.NodeVisitor

visit_Keyword(*node*)

visit_Documentation(*node*)

visit_Arguments(*node*)

visit_Tags(*node*)

visit_Return(*node*)

visit_Timeout(*node*)

visit_Teardown(*node*)

visit_KeywordCall(*node*)

visit_For(*node*)

visit_If(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**ForBuilder**(*parent*)
Bases: ast.NodeVisitor

build(*node*)

visit_KeywordCall(*node*)

visit_TemplateArguments(*node*)

visit_For(*node*)

visit_If(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.running.builder.transformers.**IfBuilder**(*parent*)
Bases: ast.NodeVisitor

build(*node*)

visit_KeywordCall(*node*)

visit_TemplateArguments(*node*)

visit_If(*node*)

visit_For(*node*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

robot.running.builder.transformers.**format_error**(*errors*)

robot.running.timeouts package

class robot.running.timeouts.**TestTimeout**(*timeout=None*, *variables=None*, *rpa=False*)
Bases: robot.running.timeouts._Timeout

type = 'Test'

set_keyword_timeout(*timeout_occurred*)

any_timeout_occurred()

active

get_message()

replace_variables(*variables*)

run(*runnable*, *args=None*, *kwargs=None*)

start()

time_left()

timed_out()

class robot.running.timeouts.**KeywordTimeout**(*timeout=None*, *variables=None*)
Bases: robot.running.timeouts._Timeout

active

get_message()

replace_variables(*variables*)

run(*runnable*, *args=None*, *kwargs=None*)

start()

time_left()

timed_out()

type = 'Keyword'

Submodules

robot.running.timeouts.ironpython module

robot.running.timeouts.jython module

robot.running.timeouts.posix module

class robot.running.timeouts.posix.**Timeout**(*timeout*, *error*)
Bases: object

execute(*runnable*)

robot.running.timeouts.windows module

class robot.running.timeouts.windows.**Timeout**(*timeout*, *error*)
Bases: object

execute(*runnable*)

Submodules

robot.running.bodyrunner module

class robot.running.bodyrunner.**BodyRunner**(*context*, *run=True*, *templated=False*)
Bases: object

run(*body*)

class robot.running.bodyrunner.**KeywordRunner**(*context*, *run=True*)
Bases: object

run(*step*, *name=None*)

class robot.running.bodyrunner.**IfRunner**(*context*, *run=True*, *templated=False*)
Bases: object

run(*data*)

robot.running.bodyrunner.**ForRunner**(*context*, *flavor='IN'*, *run=True*, *templated=False*)

class robot.running.bodyrunner.**ForInRunner**(*context*, *run=True*, *templated=False*)
Bases: object

flavor = 'IN'

run(*data*)

class robot.running.bodyrunner.**ForInRangeRunner**(*context*, *run=True*, *templated=False*)
Bases: *robot.running.bodyrunner.ForInRunner*

flavor = 'IN RANGE'

run(*data*)

class robot.running.bodyrunner.**ForInZipRunner**(*context*, *run=True*, *templated=False*)
Bases: *robot.running.bodyrunner.ForInRunner*

flavor = 'IN ZIP'

run(*data*)

class robot.running.bodyrunner.**ForInEnumerateRunner**(*context*, *run=True*, *templated=False*)
Bases: *robot.running.bodyrunner.ForInRunner*

flavor = 'IN ENUMERATE'

run(*data*)

robot.running.context module

class robot.running.context.**ExecutionContexts**
Bases: object

current

top

namespaces

start_suite(*suite*, *namespace*, *output*, *dry_run=False*)

end_suite()

robot.running.dynamicmethods module

robot.running.dynamicmethods.**no_dynamic_method**(**args*)

class robot.running.dynamicmethods.**GetKeywordNames**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

class robot.running.dynamicmethods.**RunKeyword**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

supports_kwargs

name

class robot.running.dynamicmethods.**GetKeywordDocumentation**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

class robot.running.dynamicmethods.**GetKeywordArguments**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

class robot.running.dynamicmethods.**GetKeywordTypes**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

class robot.running.dynamicmethods.**GetKeywordTags**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

class robot.running.dynamicmethods.**GetKeywordSource**(*lib*)
Bases: robot.running.dynamicmethods._DynamicMethod

name

robot.running.handlers module

robot.running.handlers.**Handler**(*library*, *name*, *method*)

robot.running.handlers.**DynamicHandler**(*library*, *name*, *method*, *doc*, *argspec*, *tags=None*)

robot.running.handlers.**InitHandler**(*library*, *method=None*, *docgetter=None*)

class robot.running.handlers.**EmbeddedArgumentsHandler**(*name_regexp*, *orig_handler*)
Bases: object

library

matches(*name*)

create_runner(*name*)

robot.running.handlerstore module

class robot.running.handlerstore.**HandlerStore**(*source*, *source_type*)
Bases: object

TEST_LIBRARY_TYPE = 'Test library'

TEST_CASE_FILE_TYPE = 'Test case file'

RESOURCE_FILE_TYPE = 'Resource file'

add(*handler*, *embedded=False*)

create_runner(*name*)

robot.running.importer module

class robot.running.importer.**Importer**
Bases: object

reset()

close_global_library_listeners()

import_library(*name*, *args*, *alias*, *variables*)

import_resource(*path*)

class robot.running.importer.**ImportCache**
Bases: object

Keeps track on and optionally caches imported items.

Handles paths in keys case-insensitively on case-insensitive OSes. Unlike dicts, this storage accepts mutable values in keys.

add(*key*, *item=None*)

values()

robot.running.librarykeywordrunner module

class robot.running.librarykeywordrunner.**LibraryKeywordRunner**(*handler*, *name=None*)

Bases: object

library

libname

longname

run(*kw*, *context*, *run=True*)

dry_run(*kw*, *context*)

class robot.running.librarykeywordrunner.**EmbeddedArgumentsRunner**(*handler*, *name*)

Bases: *robot.running.librarykeywordrunner.LibraryKeywordRunner*

dry_run(*kw*, *context*)

libname

library

longname

run(*kw*, *context*, *run=True*)

class robot.running.librarykeywordrunner.**RunKeywordRunner**(*handler*, *default_dry_run_keywords=False*)

Bases: *robot.running.librarykeywordrunner.LibraryKeywordRunner*

dry_run(*kw*, *context*)

libname

library

longname

run(*kw*, *context*, *run=True*)

robot.running.libraryscopes module

robot.running.libraryscopes.**LibraryScope**(*libcode*, *library*)

class robot.running.libraryscopes.**GlobalScope**(*library*)

Bases: object

is_global = True

start_suite()

end_suite()

start_test()

end_test()

class robot.running.libraryscopes.**TestSuiteScope**(*library*)
Bases: *robot.running.libraryscopes.GlobalScope*

is_global = False

start_suite()

end_suite()

end_test()

start_test()

class robot.running.libraryscopes.**TestCaseScope**(*library*)
Bases: *robot.running.libraryscopes.TestSuiteScope*

start_test()

end_test()

end_suite()

is_global = False

start_suite()

robot.running.model module

Module implementing test execution related model objects.

When tests are executed normally, these objects are created based on the test data on the file system by `TestSuiteBuilder`, but external tools can also create an executable test suite model structure directly. Regardless the approach to create it, the model is executed by calling *run()* method of the root test suite. See the *robot.running* package level documentation for more information and examples.

The most important classes defined in this module are *TestSuite*, *TestCase* and *Keyword*. When tests are executed, these objects can be inspected and modified by pre-run modifiers and listeners. The aforementioned objects are considered stable, but other objects in this module may still be changed in the future major releases.

class robot.running.model.**Body**(*parent=None*, *items=None*)
Bases: *robot.model.body.Body*

append(*item*)

clear()

count(*item*)

create

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

extend(*items*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

for_class
alias of *For*

if_class
alias of *If*

index(*item*, **start_and_end*)

insert(*index*, *item*)

keyword_class
alias of *Keyword*

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.running.model.**IfBranches**(*parent=None*, *items=None*)
Bases: *robot.model.body.IfBranches*

append(*item*)

clear()

count(*item*)

create

create_branch(**args*, ***kwargs*)

create_for(**args*, ***kwargs*)

create_if(**args*, ***kwargs*)

create_keyword(**args*, ***kwargs*)

extend(*items*)

filter(*keywords=None*, *fors=None*, *ifs=None*, *predicate=None*)
Filter body items based on type and/or custom predicate.

To include or exclude items based on types, give matching arguments `True` or `False` values. For example, to include only keywords, use `body.filter(keywords=True)` and to exclude FOR and IF constructs use `body.filter(fors=False, ifs=False)`. Including and excluding by types at the same time is not supported.

Custom `predicate` is a calleble getting each body item as an argument that must return `True/False` depending on should the item be included or not.

Selected items are returned as a list and the original body is not modified.

for_class = None

if_branch_class
alias of *IfBranch*

if_class = None

index(*item*, **start_and_end*)

insert(*index*, *item*)

keyword_class = None

pop(**index*)

classmethod register(*item_class*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

class robot.running.model.**Keyword**(*name=''*, *doc=''*, *args=()*, *assign=()*, *tags=()*, *timeout=None*, *type='KEYWORD'*, *parent=None*, *lineno=None*)

Bases: *robot.model.keyword.Keyword*

Represents a single executable keyword.

These keywords never have child keywords or messages. The actual keyword that is executed depends on the context where this model is executed.

See the base class for documentation of attributes not documented here.

lineno

source

run(*context*, *run=True*, *templated=None*)

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

args

assign

config(***attributes*)

Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

name

parent

repr_args = ('name', 'args', 'assign')

tags
Keyword tags as a *Tags* object.

teardown
Keyword teardown as a *Keyword* object.

This attribute is a `Keyword` object also when a keyword has no teardown but in that case its truth value is `False`.

Teardown can be modified by setting attributes directly:

```
keyword.teardown.name = 'Example'
keyword.teardown.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
keyword.teardown.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole teardown is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
keyword.teardown = None
```

New in Robot Framework 4.0. Earlier teardown was accessed like `keyword.keywords.teardown`.

timeout

type

visit(*visitor*)
Visitor interface entry-point.

class robot.running.model.**For**(*variables*, *flavor*, *values*, *parent=None*, *lineno=None*, *error=None*)

Bases: *robot.model.control.For*

body_class
alias of *Body*

lineno

error

source

run(*context*, *run=True*, *templated=False*)

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

body

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

flavor

id
Item id in format like `s1-t3-k1`.

See *TestSuite.id* for more information.

keywords
Deprecated since Robot Framework 4.0. Use *body* instead.

parent

repr_args = ('variables', 'flavor', 'values')

type = 'FOR'

values

variables

visit(*visitor*)

class robot.running.model.**If**(*parent=None*, *lineno=None*, *error=None*)
Bases: *robot.model.control.If*

body_class
alias of *IfBranches*

lineno

error

source

run(*context*, *run=True*, *templated=False*)

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

body

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

id
Root IF/ELSE id is always `None`.

parent

repr_args = ()

type = 'IF/ELSE ROOT'

visit(*visitor*)

class robot.running.model.**IfBranch**(*type='IF'*, *condition=None*, *parent=None*, *lineno=None*)
Bases: *robot.model.control.IfBranch*

body_class
alias of *Body*

lineno

source

ELSE = 'ELSE'

ELSE_IF = 'ELSE IF'

FOR = 'FOR'

FOR_ITERATION = 'FOR ITERATION'

IF = 'IF'

IF_ELSE_ROOT = 'IF/ELSE ROOT'

KEYWORD = 'KEYWORD'

MESSAGE = 'MESSAGE'

SETUP = 'SETUP'

TEARDOWN = 'TEARDOWN'

body

condition

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

id
Branch id omits the root IF/ELSE object from the parent id part.

parent

repr_args = ('type', 'condition')

type

visit(*visitor*)

class robot.running.model.**TestCase**(*name=''*, *doc=''*, *tags=None*, *timeout=None*, *template=None*, *lineno=None*)
Bases: *robot.model.testcase.TestCase*

Represents a single executable test case.

See the base class for documentation of attributes not documented here.

body_class
Internal usage only.

alias of *Body*

fixture_class
Internal usage only.

alias of *Keyword*

template

lineno

source

body
Test case body as a *Body* object.

config(***attributes*)
Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)
Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

id
Test case id in format like `s1-t3`.

See *TestSuite.id* for more information.

keywords
Deprecated since Robot Framework 4.0

Use *body*, *setup* or *teardown* instead.

longname
Test name prefixed with the long name of the parent suite.

name

parent

repr_args = ('name',)

setup
Test setup as a *Keyword* object.

This attribute is a `Keyword` object also when a test has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
test.setup.name = 'Example'
test.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
test.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
test.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `test.keywords.setup`.

tags
Test tags as a *Tags* object.

teardown
Test teardown as a *Keyword* object.

See *setup* for more information.

timeout

visit(*visitor*)
Visitor interface entry-point.

class robot.running.model.**TestSuite**(*name=''*, *doc=''*, *metadata=None*, *source=None*, *rpa=None*)

Bases: *robot.model.testsuite.TestSuite*

Represents a single executable test suite.

See the base class for documentation of attributes not documented here.

test_class
Internal usage only.

alias of *TestCase*

fixture_class
Internal usage only.

alias of *Keyword*

resource
ResourceFile instance containing imports, variables and keywords the suite owns. When data is parsed from the file system, this data comes from the same test case file that creates the suite.

classmethod from_file_system(**paths*, ***config*)
Create a *TestSuite* object based on the given `paths`.

`paths` are file or directory paths where to read the data from.

Internally utilizes the *TestSuiteBuilder* class and `config` can be used to configure how it is initialized.

New in Robot Framework 3.2.

classmethod from_model(*model*, *name=None*)
Create a *TestSuite* object based on the given `model`.

The model can be created by using the *get_model()* function and possibly modified by other tooling in the *robot.parsing* module.

New in Robot Framework 3.2.

configure(*randomize_suites=False*, *randomize_tests=False*, *randomize_seed=None*, ***options*)
A shortcut to configure a suite using one method call.

Can only be used with the root test suite.

Parameters

- **randomize_xxx** – Passed to *randomize()*.
- **options** – Passed to *SuiteConfigurer* that will then set suite attributes, call *filter()*, etc. as needed.

Example:

```
suite.configure(included_tags=['smoke'],
                doc='Smoke test results.')
```

Not to be confused with *config()* method that suites, tests, and keywords have to make it possible to set multiple attributes in one call.

randomize(*suites=True*, *tests=True*, *seed=None*)
Randomizes the order of suites and/or tests, recursively.

Parameters

- **suites** – Boolean controlling should suites be randomized.

- **tests** – Boolean controlling should tests be randomized.
- **seed** – Random seed. Can be given if previous random order needs to be re-created. Seed value is always shown in logs and reports.

run(*settings=None*, ***options*)

Executes the suite based based the given `settings` or `options`.

Parameters

- **settings** – *RobotSettings* object to configure test execution.
- **options** – Used to construct new *RobotSettings* object if `settings` are not given.

Returns *Result* object with information about executed suites and tests.

If `options` are used, their names are the same as long command line options except without hyphens. Some options are ignored (see below), but otherwise they have the same semantics as on the command line. Options that can be given on the command line multiple times can be passed as lists like `variable=['VAR1:value1', 'VAR2:value2']`. If such an option is used only once, it can be given also as a single string like `variable='VAR:value'`.

Additionally listener option allows passing object directly instead of listener name, e.g. `run('tests.robot', listener=Listener())`.

To capture stdout and/or stderr streams, pass open file objects in as special keyword arguments `stdout` and `stderr`, respectively.

Only options related to the actual test execution have an effect. For example, options related to selecting or modifying test cases or suites (e.g. `--include`, `--name`, `--prerunmodifier`) or creating logs and reports are silently ignored. The output XML generated as part of the execution can be configured, though. This includes disabling it with `output=None`.

Example:

```
stdout = StringIO()
result = suite.run(variable='EXAMPLE:value',
                   output='example.xml',
                   exitonfailure=True,
                   stdout=stdout)
print(result.return_code)
```

To save memory, the returned *Result* object does not have any information about the executed keywords. If that information is needed, the created output XML file needs to be read using the *ExecutionResult* factory method.

See the *package level* documentation for more examples, including how to construct executable test suites and how to create logs and reports based on the execution results.

See the *robot.run* function for a higher-level API for executing tests in files or directories.

config(***attributes*)

Configure model object with given attributes.

`obj.config(name='Example', doc='Something')` is equivalent to setting `obj.name = 'Example'` and `obj.doc = 'Something'`.

New in Robot Framework 4.0.

copy(***attributes*)

Return shallow copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.copy(name='New name')`.

See also *deepcopy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

deepcopy(***attributes*)
Return deep copy of this object.

Parameters **attributes** – Attributes to be set for the returned copy automatically. For example, `test.deepcopy(name='New name')`.

See also *copy()*. The difference between these two is the same as with the standard `copy.copy` and `copy.deepcopy` functions that these methods also use internally.

doc

filter(*included_suites=None*, *included_tests=None*, *included_tags=None*, *excluded_tags=None*)
Select test cases and remove others from this suite.

Parameters have the same semantics as `--suite`, `--test`, `--include`, and `--exclude` command line options. All of them can be given as a list of strings, or when selecting only one, as a single string.

Child suites that contain no tests after filtering are automatically removed.

Example:

```
suite.filter(included_tests=['Test 1', '* Example'],
             included_tags='priority-1')
```

has_tests

id
An automatically generated unique id.

The root suite has id `s1`, its child suites have ids `s1-s1`, `s1-s2`, ..., their child suites get ids `s1-s1-s1`, `s1-s1-s2`, ..., `s1-s2-s1`, ..., and so on.

The first test in a suite has an id like `s1-t1`, the second has an id `s1-t2`, and so on. Similarly keywords in suites (setup/teardown) and in tests get ids like `s1-k1`, `s1-t1-k1`, and `s1-s4-t2-k5`.

keywords
Deprecated since Robot Framework 4.0

Use *setup* or *teardown* instead.

longname
Suite name prefixed with the long name of the parent suite.

metadata
Free test suite metadata as a dictionary.

name
Test suite name. If not set, constructed from child suite names.

parent

remove_empty_suites(*preserve_direct_children=False*)
Removes all child suites not containing any tests, recursively.

repr_args = ('name',)

rpa

set_tags(*add=None*, *remove=None*, *persist=False*)
Add and/or remove specified tags to the tests in this suite.

Parameters

- **add** – Tags to add as a list or, if adding only one, as a single string.
- **remove** – Tags to remove as a list or as a single string. Can be given as patterns where `*` and `?` work as wildcards.
- **persist** – Add/remove specified tags also to new tests added to this suite in the future.

setup
Suite setup as a *Keyword* object.

This attribute is a `Keyword` object also when a suite has no setup but in that case its truth value is `False`.

Setup can be modified by setting attributes directly:

```
suite.setup.name = 'Example'
suite.setup.args = ('First', 'Second')
```

Alternatively the *config()* method can be used to set multiple attributes in one call:

```
suite.setup.config(name='Example', args=('First', 'Second'))
```

The easiest way to reset the whole setup is setting it to `None`. It will automatically recreate the underlying `Keyword` object:

```
suite.setup = None
```

New in Robot Framework 4.0. Earlier setup was accessed like `suite.keywords.setup`.

source

suites
Child suites as a *TestSuites* object.

teardown
Suite teardown as a *Keyword* object.

See *setup* for more information.

test_count
Number of the tests in this suite, recursively.

tests
Tests as a *TestCases* object.

visit(*visitor*)
Visitor interface entry-point.

class robot.running.model.**Variable**(*name*, *value*, *source=None*, *lineno=None*, *error=None*)
Bases: `object`

report_invalid_syntax(*message*, *level='ERROR'*)

class robot.running.model.**ResourceFile**(*doc=''*, *source=None*)
Bases: `object`

imports

keywords

variables

class robot.running.model.**UserKeyword**(*name*, *args=()*, *doc=''*, *tags=()*, *return_=None*, *timeout=None*, *lineno=None*, *parent=None*)
Bases: `object`

body
Child keywords as a *Body* object.

keywords
Deprecated since Robot Framework 4.0.

Use *body* or *teardown* instead.

teardown

tags

source

class robot.running.model.**Import**(*type*, *name*, *args=()*, *alias=None*, *source=None*, *lineno=None*)
Bases: object

ALLOWED_TYPES = ('Library', 'Resource', 'Variables')

directory

report_invalid_syntax(*message*, *level='ERROR'*)

class robot.running.model.**Imports**(*source*, *imports=None*)
Bases: *robot.model.itemlist.ItemList*

append(*item*)

clear()

count(*item*)

create(**args*, ***kwargs*)

extend(*items*)

index(*item*, **start_and_end*)

insert(*index*, *item*)

pop(**index*)

remove(*item*)

reverse()

sort()

visit(*visitor*)

library(*name*, *args=()*, *alias=None*, *lineno=None*)

resource(*path*, *lineno=None*)

variables(*path*, *args=()*, *lineno=None*)

robot.running.modelcombiner module

class robot.running.modelcombiner.**ModelCombiner**(*data*, *result*, ***priority*)
Bases: object

data

result

priority

robot.running.namespace module

class robot.running.namespace.**Namespace**(*variables*, *suite*, *resource*)
Bases: object

libraries

handle_imports()

import_resource(*name*, *overwrite=True*)

import_variables(*name*, *args*, *overwrite=False*)

import_library(*name*, *args=()*, *alias=None*, *notify=True*)

set_search_order(*new_order*)

start_test()

end_test()

start_suite()

end_suite(*suite*)

start_user_keyword()

end_user_keyword()

get_library_instance(*libname*)

get_library_instances()

reload_library(*libname_or_instance*)

get_runner(*name*)

class robot.running.namespace.**KeywordStore**(*resource*)
Bases: object

get_library(*name_or_instance*)

get_runner(*name*)

class robot.running.namespace.**KeywordRecommendationFinder**(*user_keywords*, *libraries*, *resources*)
Bases: object

recommend_similar_keywords(*name*)
Return keyword names similar to *name*.

static format_recommendations(*message*, *recommendations*)

robot.running.outputcapture module

class robot.running.outputcapture.**OutputCapturer**(*library_import=False*)
Bases: object

class robot.running.outputcapture.**PythonCapturer**(*stdout=True*)
Bases: object

release()

class robot.running.outputcapture.**JavaCapturer**(*stdout=True*)
Bases: object

release()

robot.running.randomizer module

class robot.running.randomizer.**Randomizer**(*randomize_suites=True*, *randomize_tests=True*, *seed=None*)

Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.running.runkwregister module

robot.running.signalhandler module

robot.running.status module

class robot.running.status.**Failure**
Bases: object

class robot.running.status.**Exit**(*failure_mode=False*, *error_mode=False*, *skip_teardown_mode=False*)
Bases: object

failure_occurred(*failure=None*)

error_occurred()

teardown_allowed

class robot.running.status.**SuiteStatus**(*parent=None*, *exit_on_failure_mode=False*, *exit_on_error_mode=False*, *skip_teardown_on_exit_mode=False*)
Bases: robot.running.status._ExecutionStatus

error_occurred()

failed

failure_occurred()

message

setup_executed(*failure=None*)

status

teardown_allowed

teardown_executed(*failure=None*)

class robot.running.status.**TestStatus**(*parent*, *test*, *skip_on_failure=None*, *critical_tags=None*, *rpa=False*)
Bases: robot.running.status._ExecutionStatus

test_failed(*failure*)

test_skipped(*reason*)

skip_if_needed()

error_occurred()

failed

failure_occurred()

message

setup_executed(*failure=None*)

status

teardown_allowed

teardown_executed(*failure=None*)

class robot.running.status.**TestMessage**(*status*)
Bases: robot.running.status._Message

setup_message = 'Setup failed:\n%s'

teardown_message = 'Teardown failed:\n%s'

setup_skipped_message = '%s'

teardown_skipped_message = '%s'

also_teardown_message = '%s\n\nAlso teardown failed:\n%s'

also_teardown_skip_message = 'Skipped in teardown:\n%s\n\nEarlier message:\n%s'

exit_on_fatal_message = 'Test execution stopped due to a fatal error.'

exit_on_failure_message = 'Failure occurred and exit-on-failure mode is in use.'

exit_on_error_message = 'Error occurred and exit-on-error mode is in use.'

message

class robot.running.status.**SuiteMessage**(*status*)
Bases: robot.running.status._Message

setup_message = 'Suite setup failed:\n%s'

setup_skipped_message = 'Skipped in suite setup:\n%s'

teardown_skipped_message = 'Skipped in suite teardown:\n%s'

teardown_message = 'Suite teardown failed:\n%s'

also_teardown_message = '%s\n\nAlso suite teardown failed:\n%s'

also_teardown_skip_message = 'Skipped in suite teardown:\n%s\n\nEarlier message:\n%s'

message

class robot.running.status.**ParentMessage**(*status*)
Bases: *robot.running.status.SuiteMessage*

setup_message = 'Parent suite setup failed:\n%s'

setup_skipped_message = 'Skipped in parent suite setup:\n%s'

teardown_skipped_message = 'Skipped in parent suite teardown:\n%s'

teardown_message = 'Parent suite teardown failed:\n%s'

also_teardown_message = '%s\n\nAlso parent suite teardown failed:\n%s'

also_teardown_skip_message = 'Skipped in suite teardown:\n%s\n\nEarlier message:\n%s'

message

robot.running.statusreporter module

class robot.running.statusreporter.**StatusReporter**(*data*, *result*, *context*, *run=True*)
Bases: object

robot.running.suiterunner module

class robot.running.suiterunner.**SuiteRunner**(*output*, *settings*)
Bases: *robot.model.visitor.SuiteVisitor*

start_suite(*suite*)
Called when suite starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

end_suite(*suite*)
Called when suite ends. Default implementation does nothing.

visit_test(*test*)
Implements traversing through tests.

Can be overridden to allow modifying the passed in `test` without calling *start_test()* or *end_test()* nor visiting keywords.

end_for(*for_*)
Called when FOR loop ends. Default implementation does nothing.

end_for_iteration(*iteration*)
Called when FOR loop iteration ends. Default implementation does nothing.

end_if(*if_*)
Called when IF/ELSE structure ends. Default implementation does nothing.

end_if_branch(*branch*)
Called when IF/ELSE branch ends. Default implementation does nothing.

end_keyword(*keyword*)
Called when keyword ends. Default implementation does nothing.

end_message(*msg*)
Called when message ends. Default implementation does nothing.

end_test(*test*)
Called when test ends. Default implementation does nothing.

start_for(*for_*)
Called when FOR loop starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_for_iteration(*iteration*)
Called when FOR loop iteration starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if(*if_*)
Called when IF/ELSE structure starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_if_branch(*branch*)
Called when IF/ELSE branch starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_keyword(*keyword*)
Called when keyword starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_message(*msg*)
Called when message starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

start_test(*test*)
Called when test starts. Default implementation does nothing.

Can return explicit `False` to stop visiting.

visit_for(*for_*)
Implements traversing through FOR loops.

Can be overridden to allow modifying the passed in `for_` without calling *start_for()* or *end_for()* nor visiting body.

visit_for_iteration(*iteration*)
Implements traversing through single FOR loop iteration.

This is only used with the result side model because on the running side there are no iterations.

Can be overridden to allow modifying the passed in `iteration` without calling *start_for_iteration()* or *end_for_iteration()* nor visiting body.

visit_if(*if_*)
Implements traversing through IF/ELSE structures.

Notice that `if_` does not have any data directly. Actual IF/ELSE branches are in its `body` and visited using *visit_if_branch()*.

Can be overridden to allow modifying the passed in `if_` without calling *start_if()* or *end_if()* nor visiting branches.

visit_if_branch(*branch*)
Implements traversing through single IF/ELSE branch.

Can be overridden to allow modifying the passed in `branch` without calling *start_if_branch()* or *end_if_branch()* nor visiting body.

visit_keyword(*kw*)
Implements traversing through keywords.

Can be overridden to allow modifying the passed in `kw` without calling *start_keyword()* or *end_keyword()* nor visiting child keywords.

visit_message(*msg*)
Implements visiting messages.

Can be overridden to allow modifying the passed in `msg` without calling *start_message()* or *end_message()*.

visit_suite(*suite*)
Implements traversing through suites.

Can be overridden to allow modifying the passed in `suite` without calling *start_suite()* or *end_suite()* nor visiting child suites, tests or keywords (setup and teardown) at all.

robot.running.testlibraries module

robot.running.testlibraries.**TestLibrary**(*name*, *args=None*, *variables=None*, *create_handlers=True*, *logger=<robot.output.logger.Logger object>*)

robot.running.usererrorhandler module

class robot.running.usererrorhandler.**UserErrorHandler**(*error*, *name*, *libname=None*)
Bases: object

Created if creating handlers fail – running raises DataError.

The idea is not to raise DataError at processing time and prevent all tests in affected test case file from executing. Instead UserErrorHandler is created and if it is ever run DataError is raised then.

Parameters

- **error** (robot.errors.DataError) – Occurred error.
- **name** (*str*) – Name of the affected keyword.
- **libname** (*str*) – Name of the affected library or resource.

longname

doc

shortdoc

create_runner(*name*)

run(*kw*, *context*, *run=True*)

dry_run(*kw*, *context*, *run=True*)

robot.running.userkeyword module

class robot.running.userkeyword.**UserLibrary**(*resource*, *source_type='Resource file'*)
Bases: object

TEST_CASE_FILE_TYPE = 'Test case file'

RESOURCE_FILE_TYPE = 'Resource file'

class robot.running.userkeyword.**UserKeywordHandler**(*keyword*, *libname*)
Bases: object

longname

shortdoc

create_runner(*name*)

class robot.running.userkeyword.**EmbeddedArgumentsHandler**(*keyword*, *libname*, *embedded*)
Bases: *robot.running.userkeyword.UserKeywordHandler*

matches(*name*)

create_runner(*name*)

longname

shortdoc

robot.running.userkeywordrunner module

class robot.running.userkeywordrunner.**UserKeywordRunner**(*handler*, *name=None*)
Bases: object

longname

libname

arguments

Return type robot.running.arguments.ArgumentSpec

run(*kw*, *context*, *run=True*)

dry_run(*kw*, *context*)

class robot.running.userkeywordrunner.**EmbeddedArgumentsRunner**(*handler*, *name*)
Bases: *robot.running.userkeywordrunner.UserKeywordRunner*

arguments

Return type robot.running.arguments.ArgumentSpec

dry_run(*kw*, *context*)

libname

longname

run(*kw*, *context*, *run=True*)

robot.tidypkg package

Submodules

robot.tidypkg.transformers module

class robot.tidypkg.transformers.**Cleaner**
Bases: *robot.parsing.model.visitor.ModelTransformer*

Clean up and normalize data.

Following transformations are made: 1) section headers are normalized to format *** *Section Name* *** 2) setting names are normalize in setting table and in test cases and

user keywords to format *Setting Name* or *[Setting Name]*

3) settings without values are removed

4) Empty lines after section headers and within items are removed

5) For loop declaration and end tokens are normalized to *FOR* and *END*

6) Old style for loop indent (i.e. a cell with only a ‘\’) are removed

visit_CommentSection(*section*)

visit_Section(*section*)

visit_Statement(*statement*)

visit_For(*loop*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.tidypkg.transformers.**NewlineNormalizer**(*newline*, *short_test_name_length*)
Bases: *robot.parsing.model.visitor.ModelTransformer*

Normalize new lines in test data

After this transformation, there is exactly one empty line between each section and between each test or user keyword.

visit_File(*node*)

visit_Section(*node*)

visit_CommentSection(*node*)

visit_TestCaseSection(*node*)

visit_TestCase(*node*)

visit_KeywordSection(*node*)

visit_Keyword(*node*)

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.tidypkg.transformers.**SeparatorNormalizer**(*use_pipes*, *space_count*)
Bases: *robot.parsing.model.visitor.ModelTransformer*

Make separators and indentation consistent.

visit_TestCase(*node*)

visit_Keyword(*node*)

visit_For(*node*)

visit_If(*node*)

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.tidypkg.transformers.**ColumnAligner**(*short_test_name_length*, *widths*)
Bases: *robot.parsing.model.visitor.ModelTransformer*

visit_TestCase(*node*)

visit_For(*node*)

visit_Statement(*statement*)

align_header(*statement*)

align_statement(*statement*)

widths_for_line(*line*)

should_write_content_after_name(*line_pos*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.tidypkg.transformers.**ColumnWidthCounter**
Bases: *robot.parsing.model.visitor.ModelTransformer*

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

class robot.tidypkg.transformers.**Aligner**(*short_test_name_length*, *setting_and_variable_name_length*, *pipes_mode*)
Bases: *robot.parsing.model.visitor.ModelTransformer*

visit_TestCaseSection(*section*)

visit_KeywordSection(*section*)

visit_Statement(*statement*)

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit(*node*)
Visit a node.

robot.utils package

Various generic utility functions and classes.

Utilities are mainly for internal usage, but external libraries and tools may find some of them useful. Utilities are generally stable, but absolute backwards compatibility between major versions is not guaranteed.

All utilities are exposed via the *robot.utils* package, and should be used either like:

```
from robot import utils

assert utils.Matcher('H?llo').match('Hillo')
```

or:

```
from robot.utils import Matcher

assert Matcher('H?llo').match('Hillo')
```

robot.utils.**read_rest_data**(*rstfile*)

Submodules

robot.utils.application module

class robot.utils.application.**Application**(*usage*, *name=None*, *version=None*, *arg_limits=None*, *env_options=None*, *logger=None*, ***auto_options*)

Bases: object

main(*arguments*, ***options*)

validate(*options*, *arguments*)

execute_cli(*cli_arguments*, *exit=True*)

console(*msg*)

parse_arguments(*cli_args*)

Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

execute(**arguments*, ***options*)

class robot.utils.application.**DefaultLogger**

Bases: object

info(*message*)

error(*message*)

close()

robot.utils.argumentparser module

robot.utils.argumentparser.**cmdline2list**(*args*, *escaping=False*)

class robot.utils.argumentparser.**ArgumentParser**(*usage*, *name=None*, *version=None*, *arg_limits=None*, *validator=None*, *env_options=None*, *auto_help=True*, *auto_version=True*, *auto_pythonpath=True*, *auto_argumentfile=True*)

Bases: object

Available options and tool name are read from the usage.

Tool name is got from the first row of the usage. It is either the whole row or anything before first ' – '.

parse_args(*args*)

Parse given arguments and return options and positional arguments.

Arguments must be given as a list and are typically sys.argv[1:].

Options are returned as a dictionary where long options are keys. Value is a string for those options that can be given only one time (if they are given multiple times the last value is used) or None if the option is not used at all. Value for options that can be given multiple times (denoted with '*' in the usage) is a list

which contains all the given values and is empty if options are not used. Options not taken arguments have value False when they are not set and True otherwise.

Positional arguments are returned as a list in the order they are given.

If 'check_args' is True, this method will automatically check that correct number of arguments, as parsed from the usage line, are given. If the last argument in the usage line ends with the character 's', the maximum number of arguments is infinite.

Possible errors in processing arguments are reported using DataError.

Some options have a special meaning and are handled automatically if defined in the usage and given from the command line:

–argumentfile can be used to automatically read arguments from a specified file. When –argumentfile is used, the parser always allows using it multiple times. Adding '*' to denote that is thus recommend. A special value 'stdin' can be used to read arguments from stdin instead of a file.

–pythonpath can be used to add extra path(s) to sys.path.

–help and –version automatically generate help and version messages. Version is generated based on the tool name and version – see __init__ for information how to set them. Help contains the whole usage given to __init__. Possible <VERSION> text in the usage is replaced with the given version. Both help and version are wrapped to Information exception.

class robot.utils.argumentparser.**ArgLimitValidator**(*arg_limits*)
Bases: object

class robot.utils.argumentparser.**ArgFileParser**(*options*)
Bases: object

process(*args*)

robot.utils.asserts module

Convenience functions for testing both in unit and higher levels.

Benefits:

- Integrates 100% with unittest (see example below)
- Can be easily used without unittest (using unittest.TestCase when you only need convenient asserts is not so nice)
- Saved typing and shorter lines because no need to have 'self.' before asserts. These are static functions after all so that is OK.
- All 'equals' methods (by default) report given values even if optional message given. This behavior can be controlled with the optional values argument.

Drawbacks:

- unittest is not able to filter as much non-interesting traceback away as with its own methods because AssertionErrors occur outside.

Most of the functions are copied more or less directly from unittest.TestCase which comes with the following license. Further information about unittest in general can be found from http://pyunit.sourceforge.net/. This module can be used freely in same terms as unittest.

unittest license:

```
Copyright (c) 1999-2003 Steve Purcell
This module is free software, and you may redistribute it and/or modify
it under the same terms as Python itself, so long as this copyright message
and disclaimer are retained in their original form.

IN NO EVENT SHALL THE AUTHOR BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF
THIS CODE, EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

THE AUTHOR SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE.  THE CODE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS,
AND THERE IS NO OBLIGATION WHATSOEVER TO PROVIDE MAINTENANCE,
SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
```

Examples:

```
import unittest
from robot.utils.asserts import assert_equal

class MyTests(unittest.TestCase):

    def test_old_style(self):
        self.assertEqual(1, 2, 'my msg')

    def test_new_style(self):
        assert_equal(1, 2, 'my msg')
```

Example output:

```
FF
======================================================================
FAIL: test_old_style (example.MyTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "example.py", line 7, in test_old_style
    self.assertEqual(1, 2, 'my msg')
AssertionError: my msg

======================================================================
FAIL: test_new_style (example.MyTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "example.py", line 10, in test_new_style
    assert_equal(1, 2, 'my msg')
  File "/path/to/robot/utils/asserts.py", line 181, in assert_equal
    _report_inequality_failure(first, second, msg, values, '!=')
  File "/path/to/robot/utils/asserts.py", line 229, in _report_inequality_failure
    raise AssertionError(msg)
AssertionError: my msg: 1 != 2

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=2)
```

robot.utils.asserts.**fail**(*msg=None*)

Fail test immediately with the given message.

robot.utils.asserts.**assert_false**(*expr*, *msg=None*)
Fail the test if the expression is True.

robot.utils.asserts.**assert_true**(*expr*, *msg=None*)
Fail the test unless the expression is True.

robot.utils.asserts.**assert_not_none**(*obj*, *msg=None*, *values=True*)
Fail the test if given object is None.

robot.utils.asserts.**assert_none**(*obj*, *msg=None*, *values=True*)
Fail the test if given object is not None.

robot.utils.asserts.**assert_raises**(*exc_class*, *callable_obj*, **args*, ***kwargs*)
Fail unless an exception of class exc_class is thrown by callable_obj.

callable_obj is invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If a correct exception is raised, the exception instance is returned by this method.

robot.utils.asserts.**assert_raises_with_msg**(*exc_class*, *expected_msg*, *callable_obj*, **args*, ***kwargs*)
Similar to fail_unless_raises but also checks the exception message.

robot.utils.asserts.**assert_equal**(*first*, *second*, *msg=None*, *values=True*, *formatter=None*)
Fail if given objects are unequal as determined by the '==' operator.

robot.utils.asserts.**assert_not_equal**(*first*, *second*, *msg=None*, *values=True*, *formatter=None*)
Fail if given objects are equal as determined by the '==' operator.

robot.utils.asserts.**assert_almost_equal**(*first*, *second*, *places=7*, *msg=None*, *values=True*)
Fail if the two objects are unequal after rounded to given places.

inequality is determined by object's difference rounded to the given number of decimal places (default 7) and comparing to zero. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

robot.utils.asserts.**assert_not_almost_equal**(*first*, *second*, *places=7*, *msg=None*, *values=True*)
Fail if the two objects are unequal after rounded to given places.

Equality is determined by object's difference rounded to to the given number of decimal places (default 7) and comparing to zero. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

robot.utils.charwidth module

A module to handle different character widths on the console.

Some East Asian characters have width of two on console, and combining characters themselves take no extra space.

See issue 604 [1] for more details about East Asian characters. The issue also contains *generate_wild_chars.py* script that was originally used to create *_EAST_ASIAN_WILD_CHARS* mapping. An updated version of the script is attached to issue 1096. Big thanks for xieyanbo for the script and the original patch.

Note that Python's *unicodedata* module is not used here because importing it takes several seconds on Jython.

[1] https://github.com/robotframework/robotframework/issues/604 [2] https://github.com/robotframework/robotframework/issues/1096

robot.utils.charwidth.**get_char_width**(*char*)

robot.utils.compat module

robot.utils.compat.**unwrap**(*func*)

robot.utils.compat.**unicode_to_str**(*self*)

robot.utils.compat.**py2to3**(*cls*)
Deprecated since RF 4.0. Use 'py3to2' instead.

robot.utils.compat.**py3to2**(*cls*)

robot.utils.compat.**with_metaclass**(*meta*, **bases*)
Create a base class with a metaclass.

robot.utils.compat.**isatty**(*stream*)

robot.utils.compress module

robot.utils.compress.**compress_text**(*text*)

robot.utils.connectioncache module

class robot.utils.connectioncache.**ConnectionCache**(*no_current_msg='No open connection.'*)
Bases: object

Cache for test libs to use with concurrent connections, processes, etc.

The cache stores the registered connections (or other objects) and allows switching between them using generated indices or user given aliases. This is useful with any test library where there's need for multiple concurrent connections, processes, etc.

This class can, and is, used also outside the core framework by SSHLibrary, Selenium(2)Library, etc. Backwards compatibility is thus important when doing changes.

current = None
Current active connection.

current_index

register(*connection*, *alias=None*)
Registers given connection with optional alias and returns its index.

Given connection is set to be the *current* connection.

If alias is given, it must be a string. Aliases are case and space insensitive.

The index of the first connection after initialization, and after *close_all()* or *empty_cache()*, is 1, second is 2, etc.

switch(*alias_or_index*)
Switches to the connection specified by the given alias or index.

Updates *current* and also returns its new value.

Alias is whatever was given to *register()* method and indices are returned by it. Index can be given either as an integer or as a string that can be converted to an integer. Raises an error if no connection with the given index or alias found.

get_connection(*alias_or_index=None*)
Get the connection specified by the given alias or index..

If `alias_or_index` is `None`, returns the current connection if it is active, or raises an error if it is not.

Alias is whatever was given to *register()* method and indices are returned by it. Index can be given either as an integer or as a string that can be converted to an integer. Raises an error if no connection with the given index or alias found.

close_all(*closer_method='close'*)
Closes connections using given closer method and empties cache.

If simply calling the closer method is not adequate for closing connections, clients should close connections themselves and use *empty_cache()* afterwards.

empty_cache()
Empties the connection cache.

Indexes of the new connections starts from 1 after this.

resolve_alias_or_index(*alias_or_index*)

class robot.utils.connectioncache.**NoConnection**(*message*)
Bases: `object`

raise_error()

robot.utils.dotdict module

class robot.utils.dotdict.**DotDict**(**args*, ***kwds*)
Bases: `collections.OrderedDict`

clear() → None. Remove all items from od.

copy() → a shallow copy of od

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of (key, value) pairs in od

iteritems()
od.iteritems -> an iterator over the (key, value) pairs in od

iterkeys() → an iterator over the keys in od

itervalues()
od.itervalues -> an iterator over the values in od

keys() → list of keys in od

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update(*[E]*, ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of values in od

viewitems() → a set-like object providing a view on od's items

viewkeys() → a set-like object providing a view on od's keys

viewvalues() → an object providing a view on od's values

robot.utils.encoding module

robot.utils.encoding.**console_decode**(*string*, *encoding='UTF-8'*, *force=False*)
Decodes bytes from console encoding to Unicode.

By default uses the system console encoding, but that can be configured using the *encoding* argument. In addition to the normal encodings, it is possible to use case-insensitive values *CONSOLE* and *SYSTEM* to use the system console and system encoding, respectively.

By default returns Unicode strings as-is. The *force* argument can be used on IronPython where all strings are *unicode* and caller knows decoding is needed.

robot.utils.encoding.**console_encode**(*string*, *errors='replace'*, *stream=<open file '<stdout>', mode 'w'>*)
Encodes Unicode to bytes in console or system encoding.

Determines the encoding to use based on the given stream and system configuration. On Python 3 and IronPython returns Unicode, otherwise returns bytes.

robot.utils.encoding.**system_decode**(*string*)
Decodes bytes from system (e.g. cli args or env vars) to Unicode.

Depending on the usage, at least cli args may already be Unicode.

robot.utils.encoding.**system_encode**(*string*, *errors='replace'*)
Encodes Unicode to system encoding (e.g. cli args and env vars).

Non-Unicode values are first converted to Unicode.

robot.utils.encodingsniffer module

robot.utils.encodingsniffer.**get_system_encoding**()

robot.utils.encodingsniffer.**get_console_encoding**()

robot.utils.error module

robot.utils.error.**get_error_message**()
Returns error message of the last occurred exception.

This method handles also exceptions containing unicode messages. Thus it MUST be used to get messages from all exceptions originating outside the framework.

robot.utils.error.**get_error_details**(*exclude_robot_traces=True*)
Returns error message and details of the last occurred exception.

robot.utils.error.**ErrorDetails**(*exc_info=None*, *exclude_robot_traces=True*)
This factory returns an object that wraps the last occurred exception

It has attributes *message*, *traceback* and *error*, where *message* contains type and message of the original error, *traceback* contains the traceback/stack trace and *error* contains the original error instance.

class robot.utils.error.**PythonErrorDetails**(*exc_type*, *exc_value*, *exc_traceback*, *exclude_robot_traces=True*)
Bases: robot.utils.error._ErrorDetails

message

traceback

class robot.utils.error.**JavaErrorDetails**(*exc_type*, *exc_value*, *exc_traceback*, *exclude_robot_traces=True*)
Bases: robot.utils.error._ErrorDetails

message

traceback

robot.utils.escaping module

robot.utils.escaping.**escape**(*item*)

robot.utils.escaping.**glob_escape**(*item*)

class robot.utils.escaping.**Unescaper**
Bases: object

unescape(*item*)

robot.utils.escaping.**split_from_equals**(*string*)

robot.utils.etreewrapper module

class robot.utils.etreewrapper.**ETSource**(*source*)
Bases: object

robot.utils.filereader module

class robot.utils.filereader.**FileReader**(*source*, *accept_text=False*)
Bases: object

Utility to ease reading different kind of files.

Supports different sources where to read the data:

- The source can be a path to a file, either as a string or as a `pathlib.Path` instance in Python 3. The file itself must be UTF-8 encoded.
- Alternatively the source can be an already opened file object, including a StringIO or BytesIO object. The file can contain either Unicode text or UTF-8 encoded bytes.
- The third options is giving the source as Unicode text directly. This requires setting `accept_text=True` when creating the reader.

In all cases bytes are automatically decoded to Unicode and possible BOM removed.

read()

readlines()

robot.utils.frange module

robot.utils.frange.**frange**(**args*)
Like `range()` but accepts float arguments.

robot.utils.htmlformatters module

class robot.utils.htmlformatters.**LinkFormatter**
Bases: object

format_url(*text*)

format_link(*text*)

class robot.utils.htmlformatters.**LineFormatter**
Bases: object

handles(*line*)

newline = '\n'

format(*line*)

class robot.utils.htmlformatters.**HtmlFormatter**
Bases: object

format(*text*)

class robot.utils.htmlformatters.**RulerFormatter**
Bases: robot.utils.htmlformatters._SingleLineFormatter

match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

format_line(*line*)

add(*line*)

end()

format(*lines*)

handles(*line*)

class robot.utils.htmlformatters.**HeaderFormatter**
Bases: robot.utils.htmlformatters._SingleLineFormatter

match()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

format_line(*line*)

add(*line*)

end()

format(*lines*)

handles(*line*)

class robot.utils.htmlformatters.**ParagraphFormatter**(*other_formatters*)
Bases: robot.utils.htmlformatters._Formatter

format(*lines*)

add(*line*)

end()

handles(*line*)

class robot.utils.htmlformatters.**TableFormatter**
Bases: robot.utils.htmlformatters._Formatter

format(*lines*)

add(*line*)

end()

handles(*line*)

class robot.utils.htmlformatters.**PreformattedFormatter**
Bases: robot.utils.htmlformatters._Formatter

format(*lines*)

add(*line*)

end()

handles(*line*)

class robot.utils.htmlformatters.**ListFormatter**
Bases: robot.utils.htmlformatters._Formatter

format(*lines*)

add(*line*)

end()

handles(*line*)

robot.utils.importer module

robot.utils.importer.**invalidate_import_caches**()

class robot.utils.importer.**Importer**(*type=None*, *logger=None*)
Bases: object

Utility that can import modules and classes based on names and paths.

Imported classes can optionally be instantiated automatically.

Parameters

- **type** – Type of the thing being imported. Used in error and log messages.
- **logger** – Logger to be notified about successful imports and other events. Currently only needs the info method, but other level specific methods may be needed in the future. If not given, logging is disabled.

import_class_or_module(*name_or_path*, *instantiate_with_args=None*, *return_source=False*)
Imports Python class/module or Java class based on the given name or path.

Parameters

- **name_or_path** – Name or path of the module or class to import.
- **instantiate_with_args** – When arguments are given, imported classes are automatically initialized using them.
- **return_source** – When true, returns a tuple containing the imported module or class and a path to it. By default returns only the imported module or class.

The class or module to import can be specified either as a name, in which case it must be in the module search path, or as a path to the file or directory implementing the module. See *import_class_or_module_by_path()* for more information about importing classes and modules by path.

Classes can be imported from the module search path using name like `modulename.ClassName`. If the class name and module name are same, using just `CommonName` is enough. When importing a class by a path, the class name and the module name must match.

Optional arguments to use when creating an instance are given as a list. Starting from Robot Framework 4.0, both positional and named arguments are supported (e.g. `['positional', 'name=value']`) and arguments are converted automatically based on type hints and default values.

If arguments needed when creating an instance are initially embedded into the name or path like `Example:arg1:arg2`, separate *split_args_from_name_or_path()* function can be used to split them before calling this method.

import_class_or_module_by_path(*path*, *instantiate_with_args=None*)
Import a Python module or Java class using a file system path.

Parameters

- **path** – Path to the module or class to import.
- **instantiate_with_args** – When arguments are given, imported classes are automatically initialized using them.

When importing a Python file, the path must end with `.py` and the actual file must also exist. When importing Java classes, the path must end with `.java` or `.class`. The Java class file must exist in both cases and in the former case also the source file must exist.

Use *import_class_or_module()* to support importing also using name, not only path. See the documentation of that function for more information about creating instances automatically.

class robot.utils.importer.**ByPathImporter**(*logger*)
Bases: robot.utils.importer._Importer

handles(*path*)

import_(*path*)

class robot.utils.importer.**NonDottedImporter**(*logger*)
Bases: robot.utils.importer._Importer

handles(*name*)

import_(*name*)

class robot.utils.importer.**DottedImporter**(*logger*)
Bases: robot.utils.importer._Importer

handles(*name*)

import_(*name*)

class robot.utils.importer.**NoLogger**
Bases: object

error(**args*, ***kws*)

warn(**args*, ***kws*)

info(**args*, ***kws*)

debug(**args*, ***kws*)

trace(**args*, ***kws*)

robot.utils.markuputils module

robot.utils.markuputils.**html_escape**(*text*, *linkify=True*)

robot.utils.markuputils.**xml_escape**(*text*)

robot.utils.markuputils.**html_format**(*text*)

robot.utils.markuputils.**attribute_escape**(*attr*)

robot.utils.markupwriters module

class robot.utils.markupwriters.**HtmlWriter**(*output*, *write_empty=True*, *usage=None*)
Bases: robot.utils.markupwriters._MarkupWriter

Parameters

- **output** – Either an opened, file like object, or a path to the desired output file. In the latter case, the file is created and clients should use *close()* method to close it.
- **write_empty** – Whether to write empty elements and attributes.

close()
Closes the underlying output file.

content(*content=None*, *escape=True*, *newline=False*)

element(*name*, *content=None*, *attrs=None*, *escape=True*, *newline=True*)

end(*name*, *newline=True*)

start(*name*, *attrs=None*, *newline=True*)

class robot.utils.markupwriters.**XmlWriter**(*output*, *write_empty=True*, *usage=None*)
Bases: robot.utils.markupwriters._MarkupWriter

Parameters

- **output** – Either an opened, file like object, or a path to the desired output file. In the latter case, the file is created and clients should use *close()* method to close it.
- **write_empty** – Whether to write empty elements and attributes.

element(*name*, *content=None*, *attrs=None*, *escape=True*, *newline=True*)

close()
Closes the underlying output file.

content(*content=None*, *escape=True*, *newline=False*)

end(*name*, *newline=True*)

start(*name*, *attrs=None*, *newline=True*)

class robot.utils.markupwriters.**NullMarkupWriter**(***kwargs*)
Bases: object

Null implementation of the _MarkupWriter interface.

start(***kwargs*)

content(***kwargs*)

element(***kwargs*)

end(***kwargs*)

close(***kwargs*)

robot.utils.match module

robot.utils.match.**eq**(*str1*, *str2*, *ignore=()*, *caseless=True*, *spaceless=True*)

class robot.utils.match.**Matcher**(*pattern*, *ignore=()*, *caseless=True*, *spaceless=True*, *regexp=False*)
Bases: object

match(*string*)

match_any(*strings*)

class robot.utils.match.**MultiMatcher**(*patterns=None*, *ignore=()*, *caseless=True*, *spaceless=True*, *match_if_no_patterns=False*, *regexp=False*)
Bases: object

match(*string*)

match_any(*strings*)

robot.utils.misc module

robot.utils.misc.**roundup**(*number*, *ndigits=0*, *return_type=None*)
Rounds number to the given number of digits.

Numbers equally close to a certain precision are always rounded away from zero. By default return value is float when ndigits is positive and int otherwise, but that can be controlled with return_type.

With the built-in round() rounding equally close numbers as well as the return type depends on the Python version.

robot.utils.misc.**printable_name**(*string*, *code_style=False*)
Generates and returns printable name from the given string.

Examples: 'simple' -> 'Simple' 'name with spaces' -> 'Name With Spaces' 'more spaces' -> 'More Spaces' 'Cases AND spaces' -> 'Cases AND Spaces' '' -> ''

If 'code_style' is True:

'mixedCAPSCamel' -> 'Mixed CAPS Camel' 'camelCaseName' -> 'Camel Case Name' 'under_score_name' -> 'Under Score Name' 'under_and space' -> 'Under And Space' 'miXed_CAPS_nAMe' -> 'MiXed CAPS NAMe' '' -> ''

robot.utils.misc.**plural_or_not**(*item*)

robot.utils.misc.**seq2str**(*sequence*, *quote="'"*, *sep=', '*, *lastsep=' and '*)
Returns sequence in format *'item 1', 'item 2' and 'item 3'*.

robot.utils.misc.**seq2str2**(*sequence*)
Returns sequence in format *[item 1 | item 2 | ...]*.

robot.utils.misc.**test_or_task**(*text*, *rpa=False*)
Replaces *{test}* in *text* with *test* or *task* depending on *rpa*.

robot.utils.normalizing module

robot.utils.normalizing.**normalize**(*string*, *ignore=()*, *caseless=True*, *spaceless=True*)
Normalizes given string according to given spec.

By default string is turned to lower case and all whitespace is removed. Additional characters can be removed by giving them in `ignore` list.

robot.utils.normalizing.**normalize_whitespace**(*string*)

robot.utils.normalizing.**lower**(*string*)

class robot.utils.normalizing.**NormalizedDict**(*initial=None*, *ignore=()*, *caseless=True*, *spaceless=True*)
Bases: `_abcoll.MutableMapping`

Custom dictionary implementation automatically normalizing keys.

Initialized with possible initial value and normalizing spec.

Initial values can be either a dictionary or an iterable of name/value pairs. In the latter case items are added in the given order.

Normalizing spec has exact same semantics as with the *normalize()* function.

copy()

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

robot.utils.platform module

robot.utils.recommendations module

class robot.utils.recommendations.**RecommendationFinder**(*normalizer=None*)
Bases: object

find_and_format(*name*, *candidates*, *message*, *max_matches=10*)

find(*name*, *candidates*, *max_matches=10*)
Return a list of close matches to *name* from *candidates*.

format(*message*, *recommendations=None*)
Add recommendations to the given message.

The recommendation string looks like:

```
<message> Did you mean:
    <recommendations[0]>
    <recommendations[1]>
    <recommendations[2]>
```

robot.utils.restreader module

class robot.utils.restreader.**CaptureRobotData**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: docutils.parsers.rst.directives.body.CodeBlock

run()

add_name(*node*)
Append self.options['name'] to node['names'] if it exists.

Also normalize the name string and register it as explicit target.

assert_has_content()
Throw an ERROR-level DirectiveError if the directive doesn't have contents.

debug(*message*)

directive_error(*level*, *message*)
Return a DirectiveError suitable for being thrown as an exception.

Call "raise self.directive_error(level, message)" from within a directive implementation to return one single system message at level *level*, which automatically gets the directive block and the line number added.

Preferably use the *debug*, *info*, *warning*, *error*, or *severe* wrapper methods, e.g. `self.error(message)` to generate an ERROR-level directive error.

error(*message*)

final_argument_whitespace = False

has_content = True

info(*message*)

option_spec = {'class': <function class_option>, 'name': <function unchanged>, 'numb

optional_arguments = 1

required_arguments = 0

severe(*message*)

warning(*message*)

class robot.utils.restreader.**RobotDataStorage**(*doctree*)
Bases: object

add_data(*rows*)

get_data()

has_data()

robot.utils.restreader.**read_rest_data**(*rstfile*)

robot.utils.robotenv module

robot.utils.robotenv.**get_env_var**(*name*, *default=None*)

robot.utils.robotenv.**set_env_var**(*name*, *value*)

robot.utils.robotenv.**del_env_var**(*name*)

robot.utils.robotenv.**get_env_vars**(*upper=False*)

robot.utils.robotinspect module

robot.utils.robotinspect.**is_java_init**(*init*)

robot.utils.robotinspect.**is_java_method**(*method*)

robot.utils.robotinspect.**is_init**(*method*)

robot.utils.robotio module

robot.utils.robotio.**file_writer**(*path=None*, *encoding='UTF-8'*, *newline=None*, *usage=None*)

robot.utils.robotio.**binary_file_writer**(*path=None*)

robot.utils.robotio.**create_destination_directory**(*path*, *usage=None*)

robot.utils.robotpath module

robot.utils.robotpath.**path_to_url**(*path*)

robot.utils.robotpath.**normpath**(*path*, *case_normalize=False*)
Replacement for os.path.normpath with some enhancements.

1. Convert non-Unicode paths to Unicode using the file system encoding.
2. NFC normalize Unicode paths (affects mainly OSX).
3. Optionally lower-case paths on case-insensitive file systems. That includes Windows and also OSX in default configuration.
4. Turn `c:` into `c:\` on Windows instead of keeping it as `c:`.

robot.utils.robotpath.**abspath**(*path*, *case_normalize=False*)
Replacement for os.path.abspath with some enhancements and bug fixes.

1. Non-Unicode paths are converted to Unicode using file system encoding.
2. Optionally lower-case paths on case-insensitive file systems. That includes Windows and also OSX in default configuration.
3. Turn `c:` into `c:\` on Windows instead of `c:\current\path`.

robot.utils.robotpath.**get_link_path**(*target*, *base*)
Returns a relative path to `target` from `base`.

If `base` is an existing file, then its parent directory is considered to be the base. Otherwise `base` is assumed to be a directory.

The returned path is URL encoded. On Windows returns an absolute path with `file:` prefix if the target is on a different drive.

robot.utils.robotpath.**find_file**(*path*, *basedir='.'*, *file_type=None*)

robot.utils.robottime module

robot.utils.robottime.**timestr_to_secs**(*timestr*, *round_to=3*)
Parses time like '1h 10s', '01:00:10' or '42' and returns seconds.

robot.utils.robottime.**secs_to_timestr**(*secs*, *compact=False*)
Converts time in seconds to a string representation.

Returned string is in format like '1 day 2 hours 3 minutes 4 seconds 5 milliseconds' with following rules:

- Time parts having zero value are not included (e.g. '3 minutes 4 seconds' instead of '0 days 0 hours 3 minutes 4 seconds')
- Hour part has a maximun of 23 and minutes and seconds both have 59 (e.g. '1 minute 40 seconds' instead of '100 seconds')

If compact has value 'True', short suffixes are used. (e.g. 1d 2h 3min 4s 5ms)

robot.utils.robottime.**format_time**(*timetuple_or_epochsecs*, *daysep=''*, *daytimesep=' '*, *timesep=':'*, *millissep=None*)
Returns a timestamp formatted from given time using separators.

Time can be given either as a timetuple or seconds after epoch.

Timetuple is (year, month, day, hour, min, sec[, millis]), where parts must be integers and millis is required only when millissep is not None. Notice that this is not 100% compatible with standard Python timetuples which do not have millis.

Seconds after epoch can be either an integer or a float.

robot.utils.robottime.**get_time**(*format='timestamp'*, *time_=None*)
Return the given or current time in requested format.

If time is not given, current time is used. How time is returned is is determined based on the given 'format' string as follows. Note that all checks are case insensitive.

- If 'format' contains word 'epoch' the time is returned in seconds after the unix epoch.
- If 'format' contains any of the words 'year', 'month', 'day', 'hour', 'min' or 'sec' only selected parts are returned. The order of the returned parts is always the one in previous sentence and order of words in 'format' is not significant. Parts are returned as zero padded strings (e.g. May -> '05').
- Otherwise (and by default) the time is returned as a timestamp string in format '2006-02-24 15:08:31'

robot.utils.robottime.**parse_time**(*timestr*)
Parses the time string and returns its value as seconds since epoch.

Time can be given in five different formats:

1) Numbers are interpreted as time since epoch directly. It is possible to use also ints and floats, not only strings containing numbers.
2) Valid timestamp ('YYYY-MM-DD hh:mm:ss' and 'YYYYMMDD hhmmss').
3) 'NOW' (case-insensitive) is the current local time.
4) 'UTC' (case-insensitive) is the current time in UTC.
5) Format 'NOW - 1 day' or 'UTC + 1 hour 30 min' is the current local/UTC time plus/minus the time specified with the time string.

Seconds are rounded down to avoid getting times in the future.

robot.utils.robottime.**get_timestamp**(*daysep=''*, *daytimesep=' '*, *timesep=':'*, *millissep='.'*)

robot.utils.robottime.**timestamp_to_secs**(*timestamp*, *seps=None*)

robot.utils.robottime.**secs_to_timestamp**(*secs*, *seps=None*, *millis=False*)

robot.utils.robottime.**get_elapsed_time**(*start_time*, *end_time*)
Returns the time between given timestamps in milliseconds.

robot.utils.robottime.**elapsed_time_to_string**(*elapsed*, *include_millis=True*)
Converts elapsed time in milliseconds to format 'hh:mm:ss.mil'.

If *include_millis* is True, '.mil' part is omitted.

class robot.utils.robottime.**TimestampCache**
Bases: object

get_timestamp(*daysep=''*, *daytimesep=' '*, *timesep=':'*, *millissep='.'*)

robot.utils.robottypes module

robot.utils.robottypes.**is_truthy**(*item*)
Returns *True* or *False* depending is the item considered true or not.

Validation rules:

- If the value is a string, it is considered false if it is *'FALSE'*, *'NO'*, *'OFF'*, *'0'*, *'NONE'* or *''*, case-insensitively.
- Other strings are considered true.
- Other values are handled by using the standard *bool()* function.

Designed to be used also by external test libraries that want to handle Boolean values similarly as Robot Framework itself. See also *is_falsy()*.

robot.utils.robottypes.**is_falsy**(*item*)
Opposite of *is_truthy()*.

robot.utils.robottypes2 module

robot.utils.robottypes2.**is_integer**(*item*)

robot.utils.robottypes2.**is_number**(*item*)

robot.utils.robottypes2.**is_bytes**(*item*)

robot.utils.robottypes2.**is_string**(*item*)

robot.utils.robottypes2.**is_unicode**(*item*)

robot.utils.robottypes2.**is_pathlike**(*item*)

robot.utils.robottypes2.**is_list_like**(*item*)

robot.utils.robottypes2.**is_dict_like**(*item*)

robot.utils.robottypes2.**type_name**(*item*, *capitalize=False*)

robot.utils.robottypes3 module

robot.utils.setter module

class robot.utils.setter.**setter**(*method*)
Bases: object

class robot.utils.setter.**SetterAwareType**
Bases: type

mro() → list
return a type's method resolution order

robot.utils.sortable module

class robot.utils.sortable.**Sortable**
Bases: object

Base class for sorting based self._sort_key

robot.utils.text module

robot.utils.text.**cut_long_message**(*msg*)

robot.utils.text.**format_assign_message**(*variable*, *value*, *cut_long=True*)

robot.utils.text.**cut_assign_value**(*value*)

robot.utils.text.**get_console_length**(*text*)

robot.utils.text.**pad_console_length**(*text*, *width*)

robot.utils.text.**split_args_from_name_or_path**(*name*)
Split arguments embedded to name or path like `Example:arg1:arg2`.

The separator can be either colon `:` or semicolon `;`. If both are used, the first one is considered to be the separator.

robot.utils.text.**split_tags_from_doc**(*doc*)

robot.utils.text.**getdoc**(*item*)

robot.utils.text.**getshortdoc**(*doc_or_item*, *linesep='\n'*)

robot.utils.text.**rstrip**(*string*)

robot.utils.unic module

robot.utils.unic.**unic**(*item*)

robot.utils.unic.**prepr**(*item*, *width=80*)

class robot.utils.unic.**PrettyRepr**(*indent=1*, *width=80*, *depth=None*, *stream=None*)
Bases: pprint.PrettyPrinter

Handle pretty printing operations onto a stream using a set of configured parameters.

indent Number of spaces to indent for each level of nesting.

width Attempted maximum number of columns in the output.

depth The maximum depth to print out nested structures.

stream The desired output stream. If omitted (or false), the standard output stream available at construction will be used.

format(*object*, *context*, *maxlevels*, *level*)

isreadable(*object*)

isrecursive(*object*)

pformat(*object*)

pprint(*object*)

robot.variables package

Implements storing and resolving variables.

This package is mainly for internal usage, but utilities for finding variables can be used externally as well.

robot.variables.**is_var**(*string*, *identifiers='$@&'*)
Deprecated since RF 3.2. Use is_variable instead.

robot.variables.**is_scalar_var**(*string*)
Deprecated since RF 3.2. Use is_scalar_variable instead.

robot.variables.**is_list_var**(*string*)
Deprecated since RF 3.2. Use is_list_variable instead.

robot.variables.**is_dict_var**(*string*)
Deprecated since RF 3.2. Use is_dict_variable instead.

robot.variables.**contains_var**(*string*, *identifiers='$@&'*)
Deprecated since RF 3.2. Use contains_variable instead.

Submodules

robot.variables.assigner module

class robot.variables.assigner.**VariableAssignment**(*assignment*)
Bases: object

validate_assignment()

assigner(*context*)

class robot.variables.assigner.**AssignmentValidator**
Bases: object

validate(*variable*)

class robot.variables.assigner.**VariableAssigner**(*assignment*, *context*)
Bases: object

assign(*return_value*)

robot.variables.assigner.**ReturnValueResolver**(*assignment*)

class robot.variables.assigner.**NoReturnValueResolver**
Bases: object

resolve(*return_value*)

class robot.variables.assigner.**OneReturnValueResolver**(*variable*)
Bases: object

resolve(*return_value*)

class robot.variables.assigner.**ScalarsOnlyReturnValueResolver**(*variables*)
Bases: robot.variables.assigner._MultiReturnValueResolver

resolve(*return_value*)

class robot.variables.assigner.**ScalarsAndListReturnValueResolver**(*variables*)
Bases: robot.variables.assigner._MultiReturnValueResolver

resolve(*return_value*)

robot.variables.evaluation module

robot.variables.evaluation.**evaluate_expression**(*expression*, *variable_store*, *modules=None*, *namespace=None*)

class robot.variables.evaluation.**EvaluationNamespace**(*variable_store*, *namespace*)
Bases: _abcoll.MutableMapping

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

robot.variables.filesetter module

class robot.variables.filesetter.**VariableFileSetter**(*store*)
Bases: object

set(*path_or_variables*, *args=None*, *overwrite=False*)

class robot.variables.filesetter.**YamlImporter**
Bases: object

import_variables(*path*, *args=None*)

class robot.variables.filesetter.**PythonImporter**
Bases: object

import_variables(*path*, *args=None*)

robot.variables.finders module

robot.variables.finders.**get_java_property**(*name*)

robot.variables.finders.**get_java_properties**()

class robot.variables.finders.**VariableFinder**(*variable_store*)
Bases: object

find(*variable*)

class robot.variables.finders.**StoredFinder**(*store*)
Bases: object

identifiers = '$@&'

find(*name*)

class robot.variables.finders.**NumberFinder**
Bases: object

identifiers = '$'

find(*name*)

class robot.variables.finders.**EmptyFinder**
Bases: object

identifiers = '$@&'

empty = <robot.utils.normalizing.NormalizedDict object>

find(*name*)

class robot.variables.finders.**InlinePythonFinder**(*variables*)
Bases: object

identifiers = '$@&'

find(*name*)

class robot.variables.finders.**ExtendedFinder**(*finder*)
Bases: object

identifiers = '$@&'

find(*name*)

class robot.variables.finders.**EnvironmentFinder**
Bases: object

identifiers = '%'

find(*name*)

robot.variables.notfound module

robot.variables.notfound.**variable_not_found**(*name*, *candidates*, *message=None*, *deco_braces=True*)
Raise DataError for missing variable name.

Return recommendations for similar variable names if any are found.

robot.variables.replacer module

class robot.variables.replacer.**VariableReplacer**(*variable_store*)
Bases: object

replace_list(*items*, *replace_until=None*, *ignore_errors=False*)
Replaces variables from a list of items.

If an item in a list is a @{list} variable its value is returned. Possible variables from other items are replaced using 'replace_scalar'. Result is always a list.

'replace_until' can be used to limit replacing arguments to certain index from the beginning. Used with Run Keyword variants that only want to resolve some of the arguments in the beginning and pass others to called keywords unmodified.

replace_scalar(*item*, *ignore_errors=False*)
Replaces variables from a scalar item.

If the item is not a string it is returned as is. If it is a variable, its value is returned. Otherwise possible variables are replaced with 'replace_string'. Result may be any object.

replace_string(*item*, *custom_unescaper=None*, *ignore_errors=False*)
Replaces variables from a string. Result is always a string.

Input can also be an already found VariableMatch.

robot.variables.scopes module

class robot.variables.scopes.**VariableScopes**(*settings*)
Bases: object

current

start_suite()

end_suite()

start_test()

end_test()

start_keyword()

end_keyword()

replace_list(*items*, *replace_until=None*, *ignore_errors=False*)

replace_scalar(*items*, *ignore_errors=False*)

replace_string(*string*, *custom_unescaper=None*, *ignore_errors=False*)

set_from_file(*path*, *args*, *overwrite=False*)

set_from_variable_table(*variables*, *overwrite=False*)

resolve_delayed()

set_global(*name*, *value*)

set_suite(*name*, *value*, *top=False*, *children=False*)

set_test(*name*, *value*)

set_keyword(*name*, *value*)

set_local_variable(*name*, *value*)

as_dict(*decoration=True*)

class robot.variables.scopes.**GlobalVariables**(*settings*)

Bases: *robot.variables.variables.Variables*

as_dict(*decoration=True*)

clear()

copy()

replace_list(*items*, *replace_until=None*, *ignore_errors=False*)

replace_scalar(*item*, *ignore_errors=False*)

replace_string(*item*, *custom_unescaper=None*, *ignore_errors=False*)

resolve_delayed()

set_from_file(*path_or_variables*, *args=None*, *overwrite=False*)

set_from_variable_table(*variables*, *overwrite=False*)

update(*variables*)

class robot.variables.scopes.**SetVariables**

Bases: object

start_suite()

end_suite()

start_test()

end_test()

start_keyword()

end_keyword()

set_global(*name*, *value*)

set_suite(*name*, *value*)

set_test(*name*, *value*)

set_keyword(*name*, *value*)

update(*variables*)

robot.variables.search module

robot.variables.search.**search_variable**(*string*, *identifiers='$@&%*'*, *ignore_errors=False*)

robot.variables.search.**contains_variable**(*string*, *identifiers='$@&'*)

robot.variables.search.**is_variable**(*string*, *identifiers='$@&'*)

robot.variables.search.**is_scalar_variable**(*string*)

robot.variables.search.**is_list_variable**(*string*)

robot.variables.search.**is_dict_variable**(*string*)

robot.variables.search.**is_assign**(*string*, *identifiers='$@&'*, *allow_assign_mark=False*)

robot.variables.search.**is_scalar_assign**(*string*, *allow_assign_mark=False*)

robot.variables.search.**is_list_assign**(*string*, *allow_assign_mark=False*)

robot.variables.search.**is_dict_assign**(*string*, *allow_assign_mark=False*)

class robot.variables.search.**VariableMatch**(*string*, *identifier=None*, *base=None*, *items=()*, *start=-1*, *end=-1*)

Bases: object

resolve_base(*variables*, *ignore_errors=False*)

name

before

match

after

is_variable()

is_scalar_variable()

is_list_variable()

is_dict_variable()

is_assign(*allow_assign_mark=False*)

is_scalar_assign(*allow_assign_mark=False*)

is_list_assign(*allow_assign_mark=False*)

is_dict_assign(*allow_assign_mark=False*)

class robot.variables.search.**VariableSearcher**(*identifiers*, *ignore_errors=False*)

Bases: object

search(*string*)

variable_state(*char*)

waiting_item_state(*char*)

item_state(*char*)

robot.variables.search.**unescape_variable_syntax**(*item*)

class robot.variables.search.**VariableIterator**(*string*, *identifiers='$@&%'*, *ignore_errors=False*)

Bases: object

robot.variables.store module

class robot.variables.store.**VariableStore**(*variables*)

Bases: object

resolve_delayed(*item=None*)

get(*name*, *default=<object object>*, *decorated=True*)

update(*store*)

clear()

add(*name*, *value*, *overwrite=True*, *decorated=True*)

as_dict(*decoration=True*)

robot.variables.tablesetter module

class robot.variables.tablesetter.**VariableTableSetter**(*store*)

Bases: object

set(*variables*, *overwrite=False*)

robot.variables.tablesetter.**VariableTableValue**(*value*, *name*, *error_reporter=None*)

class robot.variables.tablesetter.**VariableTableValueBase**(*values*, *error_reporter=None*)

Bases: object

resolve(*variables*)

report_error(*error*)

class robot.variables.tablesetter.**ScalarVariableTableValue**(*values*, *error_reporter=None*)

Bases: *robot.variables.tablesetter.VariableTableValueBase*

report_error(*error*)

resolve(*variables*)

class robot.variables.tablesetter.**ListVariableTableValue**(*values*, *error_reporter=None*)

Bases: *robot.variables.tablesetter.VariableTableValueBase*

report_error(*error*)

resolve(*variables*)

class robot.variables.tablesetter.**DictVariableTableValue**(*values*, *error_reporter=None*)

Bases: *robot.variables.tablesetter.VariableTableValueBase*

report_error(*error*)

resolve(*variables*)

robot.variables.variables module

class robot.variables.variables.**Variables**
Bases: object

Represents a set of variables.

Contains methods for replacing variables from list, scalars, and strings. On top of ${scalar}, @{list} and &{dict} variables, these methods handle also %{environment} variables.

resolve_delayed()

replace_list(*items*, *replace_until=None*, *ignore_errors=False*)

replace_scalar(*item*, *ignore_errors=False*)

replace_string(*item*, *custom_unescaper=None*, *ignore_errors=False*)

set_from_file(*path_or_variables*, *args=None*, *overwrite=False*)

set_from_variable_table(*variables*, *overwrite=False*)

clear()

copy()

update(*variables*)

as_dict(*decoration=True*)

4.1.2 Submodules

4.1.3 robot.errors module

Exceptions and return codes used internally.

External libraries should not used exceptions defined here.

exception robot.errors.**RobotError**(*message=''*, *details=''*)
Bases: exceptions.Exception

Base class for Robot Framework errors.

Do not raise this method but use more specific errors instead.

message

args

exception robot.errors.**FrameworkError**(*message=''*, *details=''*)
Bases: *robot.errors.RobotError*

Can be used when the core framework goes to unexpected state.

It is good to explicitly raise a FrameworkError if some framework component is used incorrectly. This is pretty much same as 'Internal Error' and should of course never happen.

args

message

exception robot.errors.**DataError**(*message=''*, *details=''*)
Bases: *robot.errors.RobotError*

Used when the provided test data is invalid.

DataErrors are not caught by keywords that run other keywords (e.g. *Run Keyword And Expect Error*).

args

message

exception robot.errors.**VariableError**(*message=''*, *details=''*)
Bases: *robot.errors.DataError*

Used when variable does not exist.

VariableErrors are caught by keywords that run other keywords (e.g. *Run Keyword And Expect Error*).

args

message

exception robot.errors.**KeywordError**(*message=''*, *details=''*)
Bases: *robot.errors.DataError*

Used when no keyword is found or there is more than one match.

KeywordErrors are caught by keywords that run other keywords (e.g. *Run Keyword And Expect Error*).

args

message

exception robot.errors.**TimeoutError**(*message=''*, *test_timeout=True*)
Bases: *robot.errors.RobotError*

Used when a test or keyword timeout occurs.

This exception is handled specially so that execution of the current test is always stopped immediately and it is not caught by keywords executing other keywords (e.g. *Run Keyword And Expect Error*).

keyword_timeout

args

message

exception robot.errors.**Information**(*message=''*, *details=''*)
Bases: *robot.errors.RobotError*

Used by argument parser with –help or –version.

args

message

exception robot.errors.**ExecutionStatus**(*message*, *test_timeout=False*, *keyword_timeout=False*, *syntax=False*, *exit=False*, *continue_on_failure=False*, *skip=False*, *return_value=None*)
Bases: *robot.errors.RobotError*

Base class for exceptions communicating status in test execution.

timeout

dont_continue

continue_on_failure

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

get_errors()

status

args

message

exception robot.errors.**ExecutionFailed**(*message*, *test_timeout=False*, *keyword_timeout=False*, *syntax=False*, *exit=False*, *continue_on_failure=False*, *skip=False*, *return_value=None*)

Bases: *robot.errors.ExecutionStatus*

Used for communicating failures in test execution.

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

get_errors()

message

status

timeout

exception robot.errors.**HandlerExecutionFailed**(*details*)

Bases: *robot.errors.ExecutionFailed*

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

get_errors()

message

status

timeout

exception robot.errors.**ExecutionFailures**(*errors*, *message=None*)

Bases: *robot.errors.ExecutionFailed*

get_errors()

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

message

status

timeout

exception robot.errors.**UserKeywordExecutionFailed**(*run_errors=None*, *teardown_errors=None*)

Bases: *robot.errors.ExecutionFailures*

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

get_errors()

message

status

timeout

exception robot.errors.**ExecutionPassed**(*message=None*, ***kwargs*)

Bases: *robot.errors.ExecutionStatus*

Base class for all exceptions communicating that execution passed.

Should not be raised directly, but more detailed exceptions used instead.

set_earlier_failures(*failures*)

earlier_failures

status

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

get_errors()

message

timeout

exception robot.errors.**PassExecution**(*message*)

Bases: *robot.errors.ExecutionPassed*

Used by 'Pass Execution' keyword.

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

earlier_failures

get_errors()

message

set_earlier_failures(*failures*)

status

timeout

exception robot.errors.**ContinueForLoop**(*message=None*, ***kwargs*)
Bases: *robot.errors.ExecutionPassed*

Used by 'Continue For Loop' keyword.

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

earlier_failures

get_errors()

message

set_earlier_failures(*failures*)

status

timeout

exception robot.errors.**ExitForLoop**(*message=None*, ***kwargs*)
Bases: *robot.errors.ExecutionPassed*

Used by 'Exit For Loop' keyword.

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

earlier_failures

get_errors()

message

set_earlier_failures(*failures*)

status

timeout

exception robot.errors.**ReturnFromKeyword**(*return_value=None*, *failures=None*)
Bases: *robot.errors.ExecutionPassed*

Used by 'Return From Keyword' keyword.

args

can_continue(*teardown=False*, *templated=False*, *dry_run=False*)

continue_on_failure

dont_continue

earlier_failures

get_errors()

message

set_earlier_failures(*failures*)

status

timeout

exception robot.errors.**RemoteError**(*message=''*, *details=''*, *fatal=False*, *continuable=False*)
Bases: *robot.errors.RobotError*

Used by Remote library to report remote errors.

args

message

4.1.4 robot.jarrunner module

4.1.5 robot.libdoc module

Module implementing the command line entry point for the Libdoc tool.

This module can be executed from the command line using the following approaches:

```
python -m robot.libdoc
python path/to/robot/libdoc.py
```

Instead of `python` it is possible to use also other Python interpreters.

This module also provides *libdoc()* and *libdoc_cli()* functions that can be used programmatically. Other code is for internal usage.

Libdoc itself is implemented in the *libdocpkg* package.

class robot.libdoc.**LibDoc**
Bases: *robot.utils.application.Application*

validate(*options*, *arguments*)

main(*args*, *name=''*, *version=''*, *format=None*, *docformat=None*, *specdocformat=None*, *quiet=False*)

console(*msg*)

execute(**arguments*, ***options*)

execute_cli(*cli_arguments*, *exit=True*)

parse_arguments(*cli_args*)
Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

robot.libdoc.**libdoc_cli**(*arguments=None*, *exit=True*)
Executes Libdoc similarly as from the command line.

Parameters

- **arguments** – Command line options and arguments as a list of strings. Starting from RF 4.0, defaults to `sys.argv[1:]` if not given.
- **exit** – If `True`, call `sys.exit` automatically. New in RF 4.0.

The *libdoc()* function may work better in programmatic usage.

Example:

```
from robot.libdoc import libdoc_cli

libdoc_cli(['--version', '1.0', 'MyLibrary.py', 'MyLibrary.html'])
```

robot.libdoc.**libdoc**(*library_or_resource*, *outfile*, *name=''*, *version=''*, *format=None*, *docformat=None*, *specdocformat=None*, *quiet=False*)

Executes Libdoc.

Parameters

- **library_or_resource** – Name or path of the library or resource file to be documented.
- **outfile** – Path path to the file where to write outputs.
- **name** – Custom name to give to the documented library or resource.
- **version** – Version to give to the documented library or resource.
- **format** – Specifies whether to generate HTML, XML or JSON output. If this options is not used, the format is got from the extension of the output file. Possible values are `'HTML'`, `'XML'`, `'JSON'` and `'LIBSPEC'`.
- **docformat** – Documentation source format. Possible values are `'ROBOT'`, `'reST'`, `'HTML'` and `'TEXT'`. The default value can be specified in library source code and the initial default is `'ROBOT'`.
- **specdocformat** – Specifies whether the keyword documentation in spec files is converted to HTML regardless of the original documentation format. Possible values are `'HTML'` (convert to HTML) and `'RAW'` (use original format). The default depends on the output format. New in Robot Framework 4.0.
- **quiet** – When true, the path of the generated output file is not printed the console. New in Robot Framework 4.0.

Arguments have same semantics as Libdoc command line options with same names. Run `libdoc --help` or consult the Libdoc section in the Robot Framework User Guide for more details.

Example:

```
from robot.libdoc import libdoc

libdoc('MyLibrary.py', 'MyLibrary.html', version='1.0')
```

4.1.6 robot.pythonpathsetter module

Module that adds directories needed by Robot to sys.path when imported.

robot.pythonpathsetter.**add_path**(*path*, *end=False*)

robot.pythonpathsetter.**remove_path**(*path*)

4.1.7 robot.rebot module

Module implementing the command line entry point for post-processing outputs.

This module can be executed from the command line using the following approaches:

```
python -m robot.rebot
python path/to/robot/rebot.py
```

Instead of `python` it is possible to use also other Python interpreters. This module is also used by the installed `rebot` start-up script.

This module also provides *rebot()* and *rebot_cli()* functions that can be used programmatically. Other code is for internal usage.

class robot.rebot.**Rebot**

Bases: *robot.run.RobotFramework*

main(*datasources*, ***options*)

console(*msg*)

execute(**arguments*, ***options*)

execute_cli(*cli_arguments*, *exit=True*)

parse_arguments(*cli_args*)

Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

validate(*options*, *arguments*)

robot.rebot.**rebot_cli**(*arguments=None*, *exit=True*)

Command line execution entry point for post-processing outputs.

Parameters

- **arguments** – Command line options and arguments as a list of strings. Starting from RF 3.1, defaults to `sys.argv[1:]` if not given.
- **exit** – If `True`, call `sys.exit` with the return code denoting execution status, otherwise just return the rc.

Entry point used when post-processing outputs from the command line, but can also be used by custom scripts. Especially useful if the script itself needs to accept same arguments as accepted by Rebot, because the script can just pass them forward directly along with the possible default values it sets itself.

Example:

```
from robot import rebot_cli

rebot_cli(['--name', 'Example', '--log', 'NONE', 'o1.xml', 'o2.xml'])
```

See also the *rebot()* function that allows setting options as keyword arguments like `name="Example"` and generally has a richer API for programmatic Rebot execution.

robot.rebot.**rebot**(**outputs*, ***options*)

Programmatic entry point for post-processing outputs.

Parameters

- **outputs** – Paths to Robot Framework output files similarly as when running the `rebot` command on the command line.

- **options** – Options to configure processing outputs. Accepted options are mostly same as normal command line options to the `rebot` command. Option names match command line option long names without hyphens so that, for example, `--name` becomes `name`.

The semantics related to passing options are exactly the same as with the *run()* function. See its documentation for more details.

Examples:

```
from robot import rebot

rebot('path/to/output.xml')
with open('stdout.txt', 'w') as stdout:
    rebot('o1.xml', 'o2.xml', name='Example', log=None, stdout=stdout)
```

Equivalent command line usage:

```
rebot path/to/output.xml
rebot --name Example --log NONE o1.xml o2.xml > stdout.txt
```

4.1.8 robot.run module

Module implementing the command line entry point for executing tests.

This module can be executed from the command line using the following approaches:

```
python -m robot.run
python path/to/robot/run.py
```

Instead of `python` it is possible to use also other Python interpreters. This module is also used by the installed `robot` start-up script.

This module also provides *run()* and *run_cli()* functions that can be used programmatically. Other code is for internal usage.

class robot.run.**RobotFramework**

Bases: *robot.utils.application.Application*

main(*datasources*, ***options*)

validate(*options*, *arguments*)

console(*msg*)

execute(**arguments*, ***options*)

execute_cli(*cli_arguments*, *exit=True*)

parse_arguments(*cli_args*)

Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

robot.run.**run_cli**(*arguments=None*, *exit=True*)

Command line execution entry point for running tests.

Parameters

- **arguments** – Command line options and arguments as a list of strings. Starting from RF 3.1, defaults to `sys.argv[1:]` if not given.
- **exit** – If `True`, call `sys.exit` with the return code denoting execution status, otherwise just return the rc.

Entry point used when running tests from the command line, but can also be used by custom scripts that execute tests. Especially useful if the script itself needs to accept same arguments as accepted by Robot Framework, because the script can just pass them forward directly along with the possible default values it sets itself.

Example:

```
from robot import run_cli

# Run tests and return the return code.
rc = run_cli(['--name', 'Example', 'tests.robot'], exit=False)

# Run tests and exit to the system automatically.
run_cli(['--name', 'Example', 'tests.robot'])
```

See also the *run()* function that allows setting options as keyword arguments like `name="Example"` and generally has a richer API for programmatic test execution.

`robot.run.run`(**tests*, ***options*)

Programmatic entry point for running tests.

Parameters

- **tests** – Paths to test case files/directories to be executed similarly as when running the `robot` command on the command line.
- **options** – Options to configure and control execution. Accepted options are mostly same as normal command line options to the `robot` command. Option names match command line option long names without hyphens so that, for example, `--name` becomes `name`.

Most options that can be given from the command line work. An exception is that options `--pythonpath`, `--argumentfile`, `--help` and `--version` are not supported.

Options that can be given on the command line multiple times can be passed as lists. For example, `include=['tag1', 'tag2']` is equivalent to `--include tag1 --include tag2`. If such options are used only once, they can be given also as a single string like `include='tag'`.

Options that accept no value can be given as Booleans. For example, `dryrun=True` is same as using the `--dryrun` option.

Options that accept string `NONE` as a special value can also be used with Python `None`. For example, using `log=None` is equivalent to `--log NONE`.

`listener`, `prerunmodifier` and `prerebotmodifier` options allow passing values as Python objects in addition to module names these command line options support. For example, `run('tests', listener=MyListener())`.

To capture the standard output and error streams, pass an open file or file-like object as special keyword arguments `stdout` and `stderr`, respectively.

A return code is returned similarly as when running on the command line. Zero means that tests were executed and no critical test failed, values up to 250 denote the number of failed critical tests, and values between 251-255 are for other statuses documented in the Robot Framework User Guide.

Example:

```
from robot import run

run('path/to/tests.robot')
run('tests.robot', include=['tag1', 'tag2'], splitlog=True)
with open('stdout.txt', 'w') as stdout:
    run('t1.robot', 't2.robot', name='Example', log=None, stdout=stdout)
```

Equivalent command line usage:

```
robot path/to/tests.robot
robot --include tag1 --include tag2 --splitlog tests.robot
robot --name Example --log NONE t1.robot t2.robot > stdout.txt
```

4.1.9 robot.testdoc module

Module implementing the command line entry point for the *Testdoc* tool.

This module can be executed from the command line using the following approaches:

```
python -m robot.testdoc
python path/to/robot/testdoc.py
```

Instead of `python` it is possible to use also other Python interpreters.

This module also provides *testdoc()* and *testdoc_cli()* functions that can be used programmatically. Other code is for internal usage.

class robot.testdoc.**TestDoc**

Bases: *robot.utils.application.Application*

main(*datasources*, *title=None*, ***options*)

console(*msg*)

execute(**arguments*, ***options*)

execute_cli(*cli_arguments*, *exit=True*)

parse_arguments(*cli_args*)

Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

validate(*options*, *arguments*)

robot.testdoc.**TestSuiteFactory**(*datasources*, ***options*)

class robot.testdoc.**TestdocModelWriter**(*output*, *suite*, *title=None*)

Bases: *robot.htmldata.htmlfilewriter.ModelWriter*

write(*line*)

write_data()

handles(*line*)

class robot.testdoc.**JsonConverter**(*output_path=None*)
Bases: object

convert(*suite*)

robot.testdoc.**testdoc_cli**(*arguments*)
Executes *Testdoc* similarly as from the command line.

Parameters **arguments** – command line arguments as a list of strings.

For programmatic usage the *testdoc()* function is typically better. It has a better API for that and does not call sys.exit() like this function.

Example:

```
from robot.testdoc import testdoc_cli

testdoc_cli(['--title', 'Test Plan', 'mytests', 'plan.html'])
```

robot.testdoc.**testdoc**(**arguments*, ***options*)
Executes *Testdoc* programmatically.

Arguments and options have same semantics, and options have same names, as arguments and options to Testdoc.

Example:

```
from robot.testdoc import testdoc

testdoc('mytests', 'plan.html', title='Test Plan')
```

4.1.10 robot.tidy module

Module implementing the command line entry point for the *Tidy* tool.

This module can be executed from the command line using the following approaches:

```
python -m robot.tidy
python path/to/robot/tidy.py
```

Instead of python it is possible to use also other Python interpreters.

This module also provides *Tidy* class and *tidy_cli()* function that can be used programmatically. Other code is for internal usage.

class robot.tidy.**Tidy**(*space_count=4*, *use_pipes=False*, *line_separator='n'*)
Bases: *robot.parsing.suitestructure.SuiteStructureVisitor*

Programmatic API for the *Tidy* tool.

Arguments accepted when creating an instance have same semantics as Tidy command line options with same names.

file(*path*, *outpath=None*)
Tidy a file.

Parameters

- **path** – Path of the input file.
- **outpath** – Path of the output file. If not given, output is returned.

Use *inplace()* to tidy files in-place.

inplace(**paths*)
Tidy file(s) in-place.

Parameters **paths** – Paths of the files to to process.

directory(*path*)
Tidy a directory.

Parameters **path** – Path of the directory to process.

All files in a directory, recursively, are processed in-place.

visit_file(*file*)

visit_directory(*directory*)

end_directory(*structure*)

start_directory(*structure*)

class robot.tidy.**TidyCommandLine**
Bases: *robot.utils.application.Application*

Command line interface for the *Tidy* tool.

Typically *tidy_cli()* is a better suited for command line style usage and *Tidy* for other programmatic usage.

main(*arguments*, *recursive=False*, *inplace=False*, *usepipes=False*, *spacecount=4*, *lineseparator='\n'*)

validate(*opts*, *args*)

console(*msg*)

execute(**arguments*, ***options*)

execute_cli(*cli_arguments*, *exit=True*)

parse_arguments(*cli_args*)
Public interface for parsing command line arguments.

Parameters **cli_args** – Command line arguments as a list

Returns options (dict), arguments (list)

Raises *Information* when –help or –version used

Raises *DataError* when parsing fails

class robot.tidy.**ArgumentValidator**
Bases: object

mode_and_args(*args*, *recursive*, *inplace*, ***others*)

line_sep(*lineseparator*, ***others*)

spacecount(*spacecount*)

robot.tidy.**tidy_cli**(*arguments*)
Executes *Tidy* similarly as from the command line.

Parameters **arguments** – Command line arguments as a list of strings.

Example:

```
from robot.tidy import tidy_cli

tidy_cli(['--spacecount', '2', 'tests.robot'])
```

4.1.11 robot.version module

robot.version.**get_version**(*naked=False*)

robot.version.**get_full_version**(*program=None*, *naked=False*)

robot.version.**get_interpreter**()

CHAPTER 5

Indices

- genindex
- modindex
- search

Python Module Index

r

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y