

# **RobotframeworkExtensions**

**v. 0.12.0**

Holger Queckenstedt

17.02.2026

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Description</b>	<b>2</b>
2.1	Keywords . . . . .	2
2.1.1	pretty_print . . . . .	2
2.1.2	normalize_path . . . . .	5
<b>3</b>	<b>Collection.py</b>	<b>7</b>
3.1	robotframeworkextensions-collection-collection . . . . .	7
3.1.1	robotframeworkextensions-collection-collection-pretty-print . . . . .	7
3.1.2	robotframeworkextensions-collection-collection-normalize-path . . . . .	8
3.1.3	robotframeworkextensions-collection-collection-get-rf-parameters . . . . .	8
3.1.4	robotframeworkextensions-collection-collection-get-parameters . . . . .	9
3.1.5	robotframeworkextensions-collection-collection-log-rf-parameter . . . . .	9
3.1.6	robotframeworkextensions-collection-collection-log-parameter . . . . .	10
3.1.7	robotframeworkextensions-collection-collection-log-rf-metadata . . . . .	11
3.1.8	robotframeworkextensions-collection-collection-log-metadata . . . . .	11
<b>4</b>	<b>Appendix</b>	<b>12</b>
<b>5</b>	<b>History</b>	<b>13</b>

# Chapter 1

## Introduction

The **robotframework-extensions-collection** extend the functionality of the Robot Framework by some useful keywords.

This covers for example string operations like normalizing a path and a pretty print method (especially for composite Python data types).

The sources of the **robotframework-extensions-collection** are available in [GitHub](#).

Informations about how to install the **robotframework-extensions-collection** can be found in the [README](#).

The **robotframework-extensions-collection** keywords are implemented in Python (as **PythonExtensionsCollection**) and the implementation can also be found in [GitHub](#).

Informations about how to install the **PythonExtensionsCollection** can be found in the [README](#).

## Chapter 2

# Description

### 2.1 Keywords

The `Collection` module of the **robotframework-extensions-collection** is the interface between the **PythonExtensionsCollection** and the RobotFramework AIO and contains the keyword definitions that can be imported in the following way:

```
Library    RobotframeworkExtensions.Collection    AS    rf.extensions
```

We recommend to use `AS` to shorten the long library name a little bit. That will make the robot code easier to read.

#### 2.1.1 `pretty_print`

The `pretty_print` keyword logs the content of parameters of any Python data type. Simple data types are logged directly. Composite data types are resolved before.

The output contains for every parameter:

- the type
- the total number of elements inside (e.g. the number of keys inside a dictionary)
- the counter number of the current element
- the value

The trace level for output is `INFO`. The output is also returned as list of strings.

### Example

The following RobotFramework AIO code defines - step by step - a parameter of composite data type (nested arrays and dictionaries) and prints the content of this with `pretty_print`:

```
set_test_variable    @{aItems1}    ${33}
...                  XYZ

set_test_variable    @{aItems}      A
...                  ${22}
...                  ${True}
...                  ${aItems1}

set_test_variable    &{dItems1}     A=${1}
...                  B=${2}

set_test_variable    &{dItems}      K1=value
...                  K2=${aItems}
...                  K3=${10}
...                  K4=${dItems1}

rf.extensions.pretty_print    ${dItems}
```

### Result

```
[DOTDICT] (4/1) > {K1} [STR] : 'value'
[DOTDICT] (4/2) > {K2} [LIST] (4/1) > [STR] : 'A'
[DOTDICT] (4/2) > {K2} [LIST] (4/2) > [INT] : 22
[DOTDICT] (4/2) > {K2} [LIST] (4/3) > [BOOL] : True
[DOTDICT] (4/2) > {K2} [LIST] (4/4) > [LIST] (2/1) > [INT] : 33
[DOTDICT] (4/2) > {K2} [LIST] (4/4) > [LIST] (2/2) > [STR] : 'XYZ'
[DOTDICT] (4/3) > {K3} [INT] : 10
[DOTDICT] (4/4) > {K4} [DOTDICT] (2/1) > {A} [INT] : 1
[DOTDICT] (4/4) > {K4} [DOTDICT] (2/2) > {B} [INT] : 2
```

Every line of output has to be interpreted strictly from left to right.

For example the meaning of the fifth line of output

```
[DOTDICT] (4/2) > {K2} [LIST] (4/4) > [LIST] (2/1) > [INT] : 33
```

is:

- The type of input parameter `dItems` is `dotdict`
- The dictionary contains 4 keys
- The current line gives information about the second key of the dictionary
- The name of the second key is `K2`
- The value of the second key is of type `list`
- The list contains 4 elements
- The current line gives information about the fourth element of the list
- The fourth element of the list is of type `list`
- The list contains 2 elements
- The current line gives information about the first element of the list
- The first element of the list is of type `int` and has the value `33`

Types are encapsulated in square brackets, counter in round brackets and key names are encapsulated in curly brackets.

**Prefix strings**

Prefix strings ( `sPrefix` ) can be used to give the lines of output a meaning, or they are used just to print also the name of the pretty printed variable ( `oData` ).

**Example**

```
rf.extensions.pretty_print    ${dItems}    PrefixString
```

**Result**

```
PrefixString : [DOTDICT] (4/1) > {K1} [STR] : 'value'
PrefixString : [DOTDICT] (4/2) > {K2} [LIST] (4/1) > [STR] : 'A'
PrefixString : [DOTDICT] (4/2) > {K2} [LIST] (4/2) > [INT] : 22
PrefixString : [DOTDICT] (4/2) > {K2} [LIST] (4/3) > [BOOL] : True
PrefixString : [DOTDICT] (4/2) > {K2} [LIST] (4/4) > [LIST] (2/1) > [INT] : 33
PrefixString : [DOTDICT] (4/2) > {K2} [LIST] (4/4) > [LIST] (2/2) > [STR] : 'XYZ'
PrefixString : [DOTDICT] (4/3) > {K3} [INT] : 10
PrefixString : [DOTDICT] (4/4) > {K4} [DOTDICT] (2/1) > {A} [INT] : 1
PrefixString : [DOTDICT] (4/4) > {K4} [DOTDICT] (2/2) > {B} [INT] : 2
```

### 2.1.2 normalize\_path

The `normalize_path` keyword normalizes local paths, paths to local network resources and internet addresses.

#### Background

It's not easy to handle paths - and especially the path separators - independent from the operating system.

Under Linux it is obvious that single slashes are used as separator within paths. Whereas the Windows explorer uses single backslashes. In both operating systems web addresses contains single slashes as separator when displayed in web browsers.

Using single backslashes within code - as content of string variables - is dangerous because the combination of a backslash and a letter can be interpreted as escape sequence - and this is maybe not the effect a user wants to have.

To avoid unwanted escape sequences backslashes have to be masked (by the usage of two of them: `"\\"`). But also this could not be the best solution because there are also applications (like the Windows explorer) that are not able to handle masked backslashes. They expect to get single backslashes within a path.

Preparing a path for best usage within code also includes collapsing redundant separators and up-level references. Python already provides functions to do this, but the outcome (path contains slashes or backslashes) depends on the operating system. And like already mentioned above also under Windows backslashes might not be the preferred choice.

It also has to be considered that redundant separators at the beginning of an address of a local network resource (like `\\server.com`) and or inside an internet address (like `https:\\server.com`) must **not** be collapsed! Unfortunately the Python function `normpath` does not consider this context.

To give the user full control about the format of a path, independent from the operating system and independent if it's a local path, a path to a local network resource or an internet address, the keyword `normalize_path` provides lot's of parameters to influence the result.

#### Example 1

Variable containing a path with:

- different types of path separators
- redundant path separators (*but backslashes have to be masked in the definition of the variable, this is **not** an unwanted redundancy*)
- up-level references

```
set_test_variable    ${sPath}    C:\\\\subfolder1\\\\.\\subfolder2\\\\\\\\\\\\\\.\\subfolder3↔
↪ \\\
```

Printing the content of `sPath` shows how the path looks like when the masking of the backslashes is resolved:

```
C:\\subfolder1\\\\.\\subfolder2\\\\\\.\\subfolder3\\
```

Usage of the `normalize_path` keyword:

```
${sPath}    rf.extensions.normalize_path    ${sPath}
```

Result (content of `sPath`):

```
C:/subfolder3
```

In case we need the Windows version (with masked backslashes instead of slashes):

```
${sPath}    rf.extensions.normalize_path    ${sPath}    bWin=${True}
```

Result (content of `sPath`):

```
C:\\subfolder3
```

The masking of backslashes can be deactivated:

```
${sPath}    rf.extensions.normalize_path    ${sPath}    bWin=${True}    bMask=${False}
```

Result (content of `sPath`):

```
C:\subfolder3
```

### Example 2

Variable containing a path of a local network resource (path starts with two masked backslashes):

```
set_test_variable    ${sPath}    \\anyserver.com\\part1//part2\\part3/part4
```

Result of normalization:

```
//anyserver.com/part1/part2/part3/part4
```

### Example 3

Variable containing an internet address:

```
set_test_variable    ${sPath}    http:\\anyserver.com\\part1//part2\\part3/part4
```

Result of normalization:

```
http://anyserver.com/part1/part2/part3/part4
```



# Chapter 3

## Collection.py

The Collection module is the interface between the PythonExtensionsCollection and the Robot Framework. This library containing the keyword definitions, can be imported in the following way:

```
Library      RobotframeworkExtensions.Collection      AS      rf.extensions
```

### 3.1 robotframeworkextensions-collection-collection

Module main class

#### 3.1.1 robotframeworkextensions-collection-collection-pretty-print

The pretty\_print keyword logs the content of parameters of any Python data type (input: oData). Simple data types are logged directly. Composite data types are resolved before.

The output contains for every parameter:

- the type
- the total number of elements inside (e.g. the number of keys inside a dictionary)
- the counter number of the current element
- the value

The trace level for output is INFO.

The output is also returned as list of strings.

#### Arguments:

- oData  
/ *Condition*: required / *Type*: any Python type /  
Data to be pretty printed
- sPrefix  
/ *Condition*: optional / *Type*: str / *Default*: None /  
If not None, this prefix string is added to every output line.

#### Returns:

- listOutLines (*list*)  
/ *Type*: list /  
List of strings containing the resolved data structure of oData (same content as printed to console).

### 3.1.2 robotframeworkextensions-collection-collection-normalize-path

The `normalize_path` keyword normalizes local paths, paths to local network resources and internet addresses

#### Arguments:

- `sPath`  
/ *Condition*: required / *Type*: str /  
The path to be normalized
- `bWin`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If `True` then the returned path contains masked backslashes as separator, otherwise slashes
- `sReferencePathAbs`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
In case of `sPath` is relative and `sReferencePathAbs` (expected to be absolute) is given, then the returned absolute path is a join of both input paths
- `bConsiderBlanks`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
If `True` then the returned path is encapsulated in quotes - in case of the path contains blanks
- `bExpandEnvVars`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
If `True` then in the returned path environment variables are resolved, otherwise not.
- `bMask`  
/ *Condition*: optional / *Type*: bool / *Default*: True (requires `bWin=True`) /  
If `bWin` is `True` and `bMask` is `True` then the returned path contains masked backslashes as separator.  
If `bWin` is `True` and `bMask` is `False` then the returned path contains single backslashes only - this might be required for applications, that are not able to handle masked backslashes.  
In case of `bWin` is `False` `bMask` has no effect.

#### Returns:

- `sPath`  
/ *Type*: str /  
The normalized path (is `None` in case of `sPath` is `None`)

### 3.1.3 robotframeworkextensions-collection-collection-get-rf-parameters

the static method `get_rf_parameters` generates a dump of all Robot Framework parameters in the current scope (including the global parameters).

The output can be filtered (to limit the dumped parameters to the desired ones).

This method returns a dictionary containing all dumped parameters.

In this module the method `get_rf_parameters` is used by the keyword `get_parameters`. `get_rf_parameters` is made a separate static method to enable also other Python modules to use this method for their own needs.

All input filter parameters are explained in detail here: [PythonExtensionsCollection.pdf](#) (section 'String operations with CString', method 'StringFilter'). Notice: In the context of this method, the filter parameter of the **PythonExtensionsCollection** have been renamed to a decapitalized format without type prefix.

The actual names of all input filter parameters are: `casesensitive`, `startswith`, `endswith`, `startswithnotwith`, `endsnotwith`, `contains`, `containsnot`, `inclregex`, `exclregex`

Further input parameters are:

- `headline`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
An additional headline logged before the parameter dump.
- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
Flag to control the console output.

### 3.1.4 robotframeworkextensions-collection-collection-get-parameters

This keyword generates a dump of all Robot Framework parameters in the current scope (including the global parameters).

The output can be filtered (to limit the dumped parameters to the desired ones).

This keyword returns a dictionary containing all dumped parameters.

All input filter parameters are explained in detail here: [PythonExtensionsCollection.pdf](#) (section 'String operations with CString', method 'StringFilter'). Notice: In the context of this keyword, the filter parameter of the **PythonExtensionsCollection** have been renamed to a decapitalized format without type prefix.

The actual names of all input filter parameters are: `casesensitive`, `startswith`, `endswith`, `startswith`, `endsnotwith`, `contains`, `containsnot`, `inclregex`, `exclregex`

Further input parameters are:

- `headline`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
An additional headline logged before the parameter dump.
- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: False /  
Flag to control the console output.

The implementation is based on the corresponding static method `get_rf_parameters` of this class.

### 3.1.5 robotframeworkextensions-collection-collection-log-rf-parameter

The static method `log_rf_parameter` logs the value of a parameter in table format. The intention behind this is a better readability of log files.

- Simple data types like strings or integers are logged in a single table line.
- Lists are logged with a separate line for every list element.
- Dictionaries are logged with a separate line for every key.
- Lists and dictionaries are resolved at top level only, not recursively. Every parameter value is logged in string format.
- Every table line can be tagged with a prefix string (optional).
- Every table can get a headline (optional).

It is also possible to control the log level (default is "INFO") and if the table shall additionally be written to console or not (default is True).

**Arguments:**

- `parameter_value`  
/ *Condition*: required / *Type*: all supported /  
The value to be logged
- `prefix`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
A prefix string added to the beginning of every line. This can be used to provide the name of the actual parameter whose value is to be logged (unfortunately this must be done manually because Python does not support this).
- `headline`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
An additional headline logged before the table.
- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Flag to control the console output.

**Returns:**

(no return values)

### 3.1.6 robotframeworkextensions-collection-collection-log-parameter

The `log_parameter` keyword logs the value of a parameter in table format. The intention behind this is a better readability of log files.

- Simple data types like strings or integers are logged in a single table line.
- Lists are logged with a separate line for every list element.
- Dictionaries are logged with a separate line for every key.
- Lists and dictionaries are resolved at top level only, not recursively. Every parameter value is logged in string format.
- Every table line can be tagged with a prefix string (optional).
- Every table can get a headline (optional).

It is also possible to control the log level (default is "INFO") and if the table shall additionally be written to console or not (default is True).

**Arguments:**

- `parameter_value`  
/ *Condition*: required / *Type*: all supported /  
The value to be logged
- `prefix`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
A prefix string added to the beginning of every line. This can be used to provide the name of the actual parameter whose value is to be logged (unfortunately this must be done manually because Python does not support this).

- `headline`  
/ *Condition*: optional / *Type*: str / *Default*: None /  
An additional headline logged before the table.
- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Flag to control the console output.

**Returns:**

(no return values)

The implementation is based on the corresponding static method `log_rf_parameter` of this class.

### 3.1.7 robotframeworkextensions-collection-collection-log-rf-metadata

The static method `log_rf_metadata` logs the value of Robot Framework metadata in table format. The intention behind this is a better readability of log files.

**Arguments:**

- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Flag to control the console output.

**Returns:**

(no return values)

### 3.1.8 robotframeworkextensions-collection-collection-log-metadata

The `log_metadata` keyword logs the value of a metadata parameter in table format. The intention behind this is a better readability of log files.

**Arguments:**

- `level`  
/ *Condition*: optional / *Type*: str / *Default*: "INFO" /  
The log level of output.
- `console`  
/ *Condition*: optional / *Type*: bool / *Default*: True /  
Flag to control the console output.

**Returns:**

(no return values)

The implementation is based on the corresponding static method `log_rf_metadata` of this class.

# Chapter 4

# Appendix

## About robotframework-extensions-collection:

### *Author*

Holger Queckenstedt ([Holger.Queckenstedt@de.bosch.com](mailto:Holger.Queckenstedt@de.bosch.com))

### *Version*

0.12.0 (17.02.2026)

### *Short Description*

Additional Robot Framework keywords

### *Homepage*

[robotframework-extensions-collection](#)

### *Documentation*

[Readme](#)  
[Main Documentation](#)

### *Sources*

[Repository](#)

### *Issues*

[Issues](#)

### *Python required*

`>=3.11`

### *License*

Apache-2.0

## Chapter 5

# History

<b>0.1.0</b>	01/2022
<i>Initial version</i>	
<b>0.2.0</b>	03/2022
<i>Setup maintenance</i>	
<b>0.3.0</b>	05/2022
<i>Documentation tool chain switched to GenPackageDoc</i>	
<b>0.4.0</b>	24.05.2022
<i>- Documentation rebuild with GenPackageDoc v. 0.13.0 - Code maintenance</i>	
<b>0.5.0</b>	31.05.2022
<i>Adapted to GenPackageDoc v. 0.15.0</i>	
<b>0.6.0</b>	02.06.2022
<i>- Documentation rebuild with GenPackageDoc v. 0.16.0 - Code maintenance</i>	
<b>0.7.0</b>	28.06.2022
<i>PythonExtensionsCollection updated to version 0.8.0</i>	
<b>0.8.0</b>	27.07.2022
<i>History reworked (requires GenPackageDoc v. 0.26.0 at least)</i>	
<b>0.9.0</b>	29.03.2023
<i>Prefix parameter added to keyword pretty_print</i>	
<b>0.10.0</b>	06.04.2023
<i>Added logging of pretty_print output to console</i>	
<b>0.11.0</b>	16.02.2026
<i>- Added keyword get_parameters to generate a dump of Robot Framework parameters in the current scope - Added keyword log_parameter to log the value of a parameter in table format, optionally with a headline - Added keyword log_metadata to log current metadata in table format</i>	
<b>0.12.0</b>	17.02.2026
<i>Usage of Setuptools replaced by usage of PIP/TOML/Setuptools</i>	

---

**RobotframeworkExtensions.pdf**

*Created at 24.06.2026 - 05:31:42*

*by GenPackageDoc v. 0.44.0 / 05.06.2026*

---