RobotframeworkExtensions

v. 0.8.4

Holger Queckenstedt

21.11.2022

Contents

Chapter 1

Introduction

The **RobotframeworkExtensions** 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 **RobotframeworkExtensions** are available in GitHub.

Informations about how to install the **RobotframeworkExtensions** can be found in the README.

The **RobotframeworkExtensions** 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 **RobotframeworkExtensions** 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    WITH NAME    rf.extensions
```

We recommend to use `WITH NAME` 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.

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    WITH NAME    rf.extensions
```

3.1 Class: Collection

Imported by:

```
from RobotframeworkExtensions.Collection import Collection
```

Module main class

3.1.1 Method: 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

Returns:

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

3.1.2 Method: 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`)

Chapter 4

Appendix

About this package:

Table 4.1: Package setup

Setup parameter	Value
Name	RobotframeworkExtensions
Version	0.8.4
Date	21.11.2022
Description	Additional Robot Framework keywords
Package URL	robotframework-extensions-collection
Author	Holger Queckenstedt
Email	Holger.Queckenstedt@de.bosch.com
Language	Programming Language :: Python :: 3
License	License :: OSI Approved :: Apache Software License
OS	Operating System :: OS Independent
Python required	>=3.0
Development status	Development Status :: 4 - Beta
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 5

History

0.1.0	01/2022
Initial version	
0.2.0	03/2022
Setup maintenance	
0.3.0	05/2022
Documentation tool chain switched to `GenPackageDoc`	
0.4.0	24.05.2022
- Documentation rebuild with `GenPackageDoc` v. 0.13.0 *- Code maintenance*	
0.5.0	31.05.2022
Adapted to `GenPackageDoc` v. 0.15.0	
0.6.0	02.06.2022
- Documentation rebuild with `GenPackageDoc` v. 0.16.0 *- Code maintenance*	
0.7.0	28.06.2022
`PythonExtensionsCollection` updated to version 0.8.0	
0.8.0	27.07.2022
History reworked (requires `GenPackageDoc` v. 0.26.0 at least)	

RobotframeworkExtensions.pdf

Created at 21.11.2022 - 10:53:59

by GenPackageDoc v. 0.36.0