Specification of Robot Framework at Bosch

Thomas Pollerspöck

Nguyen Huynh Tri Cuong
Tran Duy Ngoan
Mai Dinh Nam Son
Tran Hoang Nguyen

June 2021

! Information !

Dear reader,

Information to users.

Contents

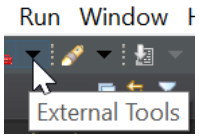
Todo list

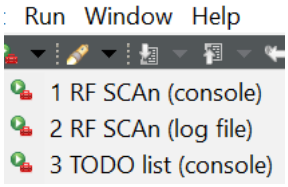
1. Code analysis

The Robotframework AIO installation provides a static code analysis to detect potential errors and violations to coding conventions. The name of the analyser is "Robocop".

Start Eclipse and select a file or a folder in Project Explorer. A missing selection causes an error.

The External Tools menu provides three preconfigured settings to start the analysis (the numbering within the context menu depends on the overall number of External Tools within your Eclipse configuration).





RF SCAn is the abbreviation for "**R**obot **F**ramework **S**tatic **C**ode **An**alysis".

1. **RF SCAn (console)**

   Output printed to console

2. **RF SCAn (log file)**

   Output printed to log file: `%ROBOTLOGPATH%\Robocop.log`

3. **TODO list (console)**

   It is possible to use the string `TODO` in code comments to indicate that still something is *todo*. But before a release all todo's should be resolved (except there is a good and documented reason to keep the marker). The static code analyser can be used to provide a list of all positions within your code, at which still such a `TODO` marker is present.

Hint: The console displays a warning regarding rule '0906'. This warning can be ignored.

2. Library

Document for the library

3. Apertis Pro

3.1 How to use DLT

3.1.1 dlt-daemon on Apertis

Verify that **dlt-daemon** has installed on Apertis Pro target or not.
`systemctl status dlt-daemon`

In case **dlt-daemon** is not available, follow below steps to install and start dlt-daemon service:

- Install **dlt-daemon** package
  `sudo apt install dlt-daemon`
- Start **dlt-daemon** service
  `sudo systemctl start dlt-daemon`

3.1.2 Apertis Pro firewall configuration

In order to capture DLT log/trace from DLT client(**DLT Viewer**, **DLTConnector**), DLT client has to comminucate with Apertis Pro (TCP/IP protocol) via port **3490** (as default).
So that, this connection should be allowed on Apertis Pro target.

Adopt settings of firewall at Apertis Pro:

- Add new rule to allow DLT service at port **3490** (as default)
  Edit `/etc/iptables/rules.v4` file to add below line

```
...
# Accept dlt for development
-A INPUT -p tcp -m state --state NEW -m tcp --dport 3490 -j ACCEPT
...
```

- Restart the firewall with changed parameters
  `sudo systemctl restart iptables.service`

3.1.3 DLTSelfTestApp

DLTSelfTestApp is an application which will be run on the Apertis Pro target for testing the DLT connection between Robotframework AIO and target.
This package is a part of Robotframework Selftest helpers.
To install **DLTSelfTestApp**, download its debian package on Apertis Pro target then execute the below command.
`sudo dpkg -i <path/to/dltselftestapp_1.0.0_amd64.deb>`

DLTSelfTestApp application will be installed in `/opt/bosch/robfw/dlt` directory and can be started with below command:
`/opt/bosch/robfw/dlt/DLTSelfTestApp`

Welcome log message `Welcome to Robotframework AIO DLTSelfTestApp...` will be sent at application startup.
Then the ping log `ping message from Robotframework AIO DLTSelfTestApp` every 5 seconds.

DLT command injection:
To perform the DLT command injection, use below information:

- App ID: **RBFW**
- Context ID: **TEST**
- Service ID: **0x1000**
- Data as Textdata

DLT log reponse of **DLTSelfTestApp** will bases on injected command:

- `welcome` : DLT reponse as above welcome message.
- `exit` : DLT reponse as `Bye...` then **DLTSelfTestApp** will be terminated.
- Other commands: DLT reponse as combination of data and string.
  e.g: `Data: 000000: 77 65 6c 63 6f 6d 65 31 32 31 32 00 xx xx xx xx welcome1212`

3.1.4 QConnectDLTLibrary

QConnectDLTLibrary is part of Robotframework AIO.
It provides the ability for handling connection to Diagnostic Log and Trace(DLT) Module. The library support for getting trace message and sending trace command

Sample Robotframework testcase which are using **QConnectDLTLibrary** to test DLTSelfTestApp on Apertis Pro target:

- Robot `Settings` (Setup, Teardown) and used `Variables` , `Keyword`

```
*** Settings ***
Documentation  This is selftest for DLT connection with DLTSelfTestApp
Library     QConnectionLibrary.ConnectionManager
Suite Setup     Open Connection
Suite Teardown  Close Connection

*** Variables ***
${CONNECTION_NAME}  TEST_CONN_DLTSelfTestApp
${DLT_CONNECTION_CONFIG} =  SEPARATOR=
...  {
...      "gen3flex@DLTLSIMWFH": {
...            "target_ip": "127.0.0.1",
...            "target_port": 4490,
...            "mode": 0,
...            "ecu": "ECU1",
...            "com_port": "COM1",
...            "baudrate": 115200,
...            "server_ip": "localhost",
...            "server_port": 1234
...      }
...  }

*** Keywords ***
Close Connection
   disconnect  ${CONNECTION_NAME}
   Log to console    \nDLT connection has been closed!

Open Connection
   ${dlt_config} =    evaluate    json.loads('''${DLT_CONNECTION_CONFIG}''')    json

   connect             conn_name=${CONNECTION_NAME}
   ...                 conn_type=DLT
   ...                 conn_mode=dltconnector
   ...                 conn_conf=${dlt_config}

   Log to console    \nDLT connection has been opened successfully!
```

- Sample robot testcase to verify the ping message from DLTSelfTestApp

```
*** Test Cases ***
Match log/trace from DLTSelfTestApp
   [Documentation]   Match log/trace from DLTSelfTestApp
   [Tags]   DLTSelfTestApp
   ${res}=    verify     conn_name=${CONNECTION_NAME}
   ...                   search_pattern=(DLT:0x01.*RBFW.*)
   ...                   timeout=6    # DLTSelfTestApp pings a message every 5 seconds

   # log to console     \n${res}[0]
   # verify that reponse message should contain "Ping" keyword
   Should Match Regexp     ${res}[0]    DLT:0x01.*RBFW.*Ping.*
```

- Sample robot testcase to verify command injection with DLTSelfTestApp

```
Command injection with DLTSelfTestApp
   [Documentation]   Get log/trace from DLTSelfTestApp
   [Tags]   DLTSelfTestApp
   ${res}=    verify     conn_name=${CONNECTION_NAME}
   ...                   search_pattern=(DLT:0x01.*RBFW.*Welcome.*)
   ...                   send_cmd=DLT_CALL_SW_INJECTION_ECU ECU1 1000 RBFW TEST 'welcome'

   # log to console     \n${res}[0]

   ${res}=    verify     conn_name=${CONNECTION_NAME}
   ...                   search_pattern=(DLT:0x01.*RBFW.*other_cmd.*)
   ...                   send_cmd=DLT_CALL_SW_INJECTION_ECU ECU1 1000 RBFW TEST 'other_cmd'

   # log to console     \n${res}[0]
```

Please refer QConnectDLTLibrary repository for more details about usage and other example testcase for DLT commection.

4. Additional libraries

4.1 PythonExtensionsCollection

PythonExtensionsCollection

v. 0.7.1

Holger Queckenstedt

14.06.2022

Contents

A

Chapter 1

Introduction

The Python Extensions Collection package extends the functionality of Python by some useful functions that are not available in Python immediately.

This covers for example string operations like normalizing a path or searching for parent directories within a path.

The Python Extensions Collection contains several Python modules and every module has to be imported separately in case of the functions inside are needed.

Module import

The modules of the Python Extension Collection and their methods can be accessed in the following ways:

CFile

```
from PythonExtensionsCollection.File.CFile import CFile
...
sFile = r"%TMP%\File.txt"
oFile = CFile(sFile)
```

Please consider that `oFile` is an instance of the class `CFile` - *and not a file handle.*

CString

```
from PythonExtensionsCollection.String.CString import CString
...
sPath = CString.NormalizePath(sPath)
...
bAck = CString.StringFilter(sString, ...)
...
sResult = CString.FormatResult(sMethod="", bSuccess=True, sResult="")
```

CUtils

```
from PythonExtensionsCollection.Utils.CUtils import *
...
PrettyPrint(oData)
```

Chapter 2

Description

2.1 Path normalization

It's not easy to handle paths - and especially the path separators - independend 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, independend from the operating system and independend if it's a local path, a path to a local network resource or an internet address, the function `CString::NormalizePath()` provides lot's of parameters to influence the result.

2.2 File access with CFile

The motivation for the CFile module contains two main topics:

1. More user control by introducing further parameter for file access functions. With high priority `CFile` enables the user to take care about that nothing existing is overwritten accidently.
2. Hide the file handles und use the mechanism of class variables to avoid access violations independend from the way different operation systems like Windows and Unix are handling this.

This shortens the code, eases the implementation and makes tests (in which this module is used) more stable.

Define two variables with path and name of test files.

Under Windows:

```
sFile_1 = r"%TMP%\CFile_TestFile_1.txt"
sFile_2 = r"%TMP%\CFile_TestFile_2.txt"
```

Or under Linux:

```
sFile_1 = r"/tmp/CFile_TestFile_1.txt"
sFile_2 = r"/tmp/CFile_TestFile_2.txt"
```

The first class instance:

```
oFile_1 = CFile(sFile_1)
```

`oFile_1` is the instance of a class - *and not the file handle.* The file handle is hidden, the user has nothing to do with it.

Every class instance can work with one single file only (during the complete instance lifetime) and has exclusive access to this file.

No other class instance is allowed to use this file. Therefore the second line in the following code throws an exception:

```
oFile_1_A = CFile(sFile_1)
oFile_1_B = CFile(sFile_1)
```

It's more save to implement in this way:

```
try:
   oFile_1 = CFile(sFile_1)
except Exception as reason:
   print(str(reason))
```

For writing content to files two methods are available: `Write()` and `Append()`.

Using `Write()` causes the class to open the file for writing ('w') - in case of the file is not already opened for writing. Using `Append()` causes the class to open the file for appending ('a') - in case of the file is not already opened for appending.

Switching between `Write()` and `Append()` causes an intermediate file handle `close()` internally!

Write some content to file:

```
bSuccess, sResult = oFile_1.Write("A B C")
print(f"> sResult oFile_1.Write : '{sResult}' / bSuccess : {bSuccess}")
```

Most of the functions returns at least `bSuccess` and `sResult`.

- `bSuccess` is `True` in case of no error occurred.
- `bSuccess` is `False` in case of an error occurred.
- `bSuccess` is `None` in case of a very fatal error occurred (exceptions).
- `sResult` contains details about what happens during computation.

It is possible now to continue with using `oFile_1.Write("...")`; the content will be appended - as long as the file is still open for writing.

Some functions close the file handle (e.g. `ReadLines()`). Therefore sequences like

```
oFile_1.Write("...")
oFile_1.Readlines("...")
oFile_1.Write("...")
```

should be avoided - because the `Write()` after the `ReadLines()` starts the file from scratch and the file content written by the previous `Write()` calls is lost.

For appending content to a file use the function `Append()`.

Append content to file:

```
bSuccess, sResult = oFile_1.Append("A B C")
```

For reading content from a file use the function `ReadLines()`.

Read from file:

```
listLines_1, bSuccess, sResult = oFile_1.ReadLines()
for sLine in listLines_1:
   print(f"{sLine}")
```

Additionally to `bSuccess` and `sResult` the function returnes a list of lines.

Internally `ReadLines()` takes care about:

- Closing the file - in case the file is still opened
- Opening the file for reading
- Reading the content line by line until the end of file is reached
- Closing the file

To avoid code like this

```
for sLine in listLines_1:
   print(f"{sLine}")
```

it is also possible to let `ReadLines()` do this:

```
listLines_1, bSuccess, sResult = oFile_1.ReadLines(bToScreen=True)
```

A function to read a single line from file only is not available, but it is possible to use some filter parameter of `ReadLines()` to reduce the amount of content already during the file is read. This prevents the user from implementing further loops.

Let's assume the following:

- The file `sFile_1` contains empty lines
- The file `sFile_1` contains also lines, that are commented out (with a hash '#' at the beginning)
- We want `ReadLines()` to skip empty lines and lines that are commented out

This can be imlemented in the following way.

Read a subset of file content:

```
listLines_1, bSuccess, sResult = oFile_1.ReadLines(bSkipBlankLines=True,
                                                   sComment='#')
```

It is a good practice to close file handles as soon as possible. Therefore `CFile` provides the possibility to do this explicitely.

Close a file handle:

```
bSuccess, sResult = oFile_1.Close()
```

This makes sense in case of later again access to this file is needed.

Additionally to that the file handle is closed implicitely:

- in case of it is required (e.g. when switching between read and write access),
- in case of the class instance is destoyed.

Therefore an alternative to the `Close()` function is the deletion of the class instance:

```
del oFile_1
```

This makes sense in case of access to this file is not needed any more.

It is recommended to prefer `del` (instead of `Close()`) to avoid to keep too much not used objects for a too long length of time in memory.

A file can be copied to another file.

Copy a file:

```
bSuccess, sResult = oFile_1.CopyTo(sFile_2)
```

The destination (`sFile_2` in the example above) can either be a full path and name of a file or the path only.

It makes a difference if the destination file exists or not. The optional parameter `bOverwrite` controls the behavior of `CopyTo()`.

The default is that it is not allowed to overwrite an existing destination file: `bOverwrite` is `False`. `CopyTo()` returns `bSuccess = False` in this case.

In case the user want to allow `CopyTo()` to overwrite existing destination files, it has to be coded explicitely:

```
bSuccess, sResult = oFile_1.CopyTo(sFile_2, bOverwrite=True)
```

A file can be moved to another file.

Move a file:

```
bSuccess, sResult = oFile_1.MoveTo(sFile_2)
```

Also `MoveTo()` supports `bOverwrite`. The behavior is the same as `CopyTo()`.

A file can be deleted.

Delete a file:

```
bSuccess, sResult = oFile_1.Delete()
```

It is possible to distinguish between two different motivations to delete a file:

1. *Explicitely do a deletion*

   This requires that the file to be deleted, does exist.

2. *Making sure only that the files does not exist*

   In this case it doesn't matter that maybe there is nothing to delete because the file already does not exist.

The optional parameter `bConfirmDelete` controls this behavior.

Default is that `Delete()` requires an existing file to delete:

```
bSuccess, sResult = oFile_1.Delete(bConfirmDelete=True)
```

In case of the file does not exist, `Delete()` returns `bSuccess = False`.

`Delete()` also returns `bSuccess = False|None` in case of an existing file cannot be deleted (e.g. because of an access violation).

If it doesn't matter it the file exists or not, it has to be coded explicitely:

```
bSuccess, sResult = oFile_1.Delete(bConfirmDelete=False)
```

In this case `Delete()` only returns `bSuccess = False|None` in case of an existing file cannot be deleted (e.g. because of an access violation).

Avoid access violations

Like already mentioned above every instance of `CFile` has an exclusive access to it's own file.

Only in case of `CopyTo()` and `MoveTo()` other files are involved: the destination files.

To avoid access violations it is not possible to copy or move a file to another file, that is under access of another instance of `CFile`.

In the following example `oFile_1.CopyTo(sFile_2)` returns `bSuccess = False` because `sFile_2` is already in access by `oFile_2`.

```
oFile_1 = CFile(sFile_1)
bSuccess, sResult = oFile_1.Write("A B C")

oFile_2 = CFile(sFile_2)
listLines_2, bSuccess, sResult = oFile_2.ReadLines()

bSuccess, sResult = oFile_1.CopyTo(sFile_2)

del oFile_1
del oFile_2
```

The solution is to delete the class instances as early as possible.

In the following example the copying is successful:

```
oFile_1 = CFile(sFile_1)
bSuccess, sResult = oFile_1.Write("A B C")

oFile_2 = CFile(sFile_2)
```

```
listLines_2, bSuccess, sResult = oFile_2.ReadLines()
del oFile_2

bSuccess, sResult = oFile_1.CopyTo(sFile_2)
del oFile_1
```

Chapter 3

CFile.py

3.1 Class: enFileStatiType

```
PythonExtensionsCollection.File.CFile
```

The class `enFileStatiType` defines the sollowing file states:

- `closed`
- `openedforwriting`
- `openedforappending`
- `openedforreading`

3.2 Class: CFile

```
PythonExtensionsCollection.File.CFile
```

The class `CFile` provides a small set of file functions with extended parametrization (like switches defining if a file is allowed to be overwritten or not).

Most of the functions at least returns `bSuccess` and `sResult`.

- `bSuccess` is `True` in case of no error occurred.
- `bSuccess` is `False` in case of an error occurred.
- `bSuccess` is `None` in case of a very fatal error occurred (exceptions).
- `sResult` contains details about what happens during computation.

Every instance of CFile handles one single file only and forces exclusive access to this file.

It is not possible to create an instance of this class with a file that is already in use by another instance.

It is also not possible to use `CopyTo` or `MoveTo` to overwrite files that are already in use by another instance. This makes the file handling more save against access violations.

3.2.1 Method: Close

Closes the opened file.

Arguments:

(no args)

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

3.2.2 Method: Delete

Deletes the current file.

Arguments:

- `bConfirmDelete`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  Defines if it will be handled as error if the file does not exist.
  If `True`: If the file does not exist, the method indicates an error (`bSuccess = False`).
  If `False`: It doesn't matter if the file exists or not.

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

3.2.3 Method: Write

Writes the content of a variable `Content` to file.

Arguments:

- `Content`
  / *Condition*: required / *Type*: one of: str, list, tuple, set, dict, dotdict /
  If `Content` is not a string, the `Write` method resolves the data structure before writing the content to file.
- `nVSpaceAfter`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  Adds vertical space `nVSpaceAfter` (= number of blank lines) after `Content`.

- `sPrefix`
  / *Condition*: optional / *Type*: str / *Default*: None /
  sPrefix is added to every line of output (in case of sPrefix is not None‘).
- `bToScreen`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  Prints `Content` also to screen (in case of `bToScreen` is `True`).

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

3.2.4 Method: Append

Appends the content of a variable `Content` to file.

Arguments:

- `Content`
  / *Condition*: required / *Type*: one of: str, list, tuple, set, dict, dotdict /
  If `Content` is not a string, the `Write` method resolves the data structure before writing the content to file.
- `nVSpaceAfter`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  Adds vertical space `nVSpaceAfter` (= number of blank lines) after `Content`.
- `sPrefix`
  / *Condition*: optional / *Type*: str / *Default*: None /
  sPrefix is added to every line of output (in case of sPrefix is not None‘).
- `bToScreen`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  Prints `Content` also to screen (in case of `bToScreen` is `True`).

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

3.2.5 Method: ReadLines

Reads content from current file. Returns an array of lines together with `bSuccess` and `sResult` (feedback).

The method takes care of opening and closing the file. The complete file content is read by `ReadLines` in one step, but with the help of further parameters it is possible to reduce the content by including and excluding lines.

The logical join of all filter is: `AND`.

Arguments:

- `bCaseSensitive`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  - If `True`, the standard filters work case sensitive, otherwise not.
  - This has no effect to the regular expression based filters `sInclRegEx` and `sExclRegEx`.
- `bSkipBlankLines`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `True`, blank lines will be skipped, otherwise not.
- `sComment`
  / *Condition*: optional / *Type*: str / *Default*: None /
  In case of a line starts with the string `sComment`, this line is skipped.
- `sStartsWith`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string starts with the string `sStartsWith`
  - More than one string can be provided (semicolon separated; logical join: `OR`)
- `sEndsWith`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string ends with the string `sEndsWith`
  - More than one string can be provided (semicolon separated; logical join: `OR`)
- `sStartsNotWith`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string starts not with the string `sStartsNotWith`
  - More than one string can be provided (semicolon separated; logical join: `AND`)
- `sEndsNotWith`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string ends not with the string `sEndsNotWith`
  - More than one string can be provided (semicolon separated; logical join: `AND`)
- `sContains`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string contains the string `sContains` at any position

  - More than one string can be provided (semicolon separated; logical join: `OR`)

- `sContainsNot`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - The criterion of this filter is fulfilled in case of the input string does **not** contain the string `sContainsNot` at any position
  - More than one string can be provided (semicolon separated; logical join: `AND`)

- `sInclRegEx`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - *Include* filter based on regular expressions (consider the syntax of regular expressions!)
  - The criterion of this filter is fulfilled in case of the regular expression `sInclRegEx` matches the input string
  - Leading and trailing blanks within the input string are considered
  - `bCaseSensitive` has no effect
  - A semicolon separated list of several regular expressions is **not** supported

- `sExclRegEx`
  / *Condition*: optional / *Type*: str / *Default*: None /
  - *Exclude* filter based on regular expressions (consider the syntax of regular expressions!)
  - The criterion of this filter is fulfilled in case of the regular expression `sExclRegEx` does **not** match the input string
  - Leading and trailing blanks within the input string are considered
  - `bCaseSensitive` has no effect
  - A semicolon separated list of several regular expressions is **not** supported

- `bLStrip`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `True`, leading spaces are removed from line before the filters are used, otherwise not.

- `bRStrip`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  If `True`, trailing spaces are removed from line before the filters are used, otherwise not.

- `bToScreen`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `True`, the content read from file is also printed to screen, otherwise not.

3.2.6 Method: GetFileInfo

Returns the following informations about the file (encapsulated within a dictionary `dFileInfo`):

Returns:

- Key `sFile`
  / *Type*: str /
  Path and name of current file

- Key `bFileIsExisting`
  / *Type*: bool /
  `True` if file is existing, otherwise `False`

- Key `sFileName`

  / *Type*: str /

  The name of the current file (incl. extension)

- Key `sFileExtension`

  / *Type*: str /

  The extension of the current file

- Key `sFileNameOnly`

  / *Type*: str /

  The pure name of the current file (without extension)

- Key `sFilePath`

  / *Type*: str /

  The the path to current file

- Key `bFilePathIsExisting`

  / *Type*: bool /

  `True` if file path is existing, otherwise `False`

3.2.7 Method: CopyTo

Copies the current file to `sDestination`, that can either be a path without file name or a path together with a file name.

In case of the destination file already exists and `bOverwrite` is `True`, than the destination file will be overwritten.

In case of the destination file already exists and `bOverwrite` is `False` (default), than the destination file will not be overwritten and `CopyTo` returns `bSuccess = False`.

Arguments:

- `sDestination`

  / *Condition*: required / *Type*: string /

  The path to destination file (either incl. file name or without file name)

- `bOverwrite`

  / *Condition*: optional / *Type*: bool / *Default*: False /

  - In case of the destination file already exists and `bOverwrite` is `True`, than the destination file will be overwritten.
  - In case of the destination file already exists and `bOverwrite` is `False` (default), than the destination file will not be overwritten and `CopyTo` returns `bSuccess = False`.

Returns:

- `bSuccess`

  / *Type*: bool /

  Indicates if the computation of the method was successful or not.

- `sResult`

  / *Type*: str /

  The result of the computation of the method.

3.2.8 Method: MoveTo

Moves the current file to `sDestination`, that can either be a path without file name or a path together with a file name.

Arguments:

- `sDestination`

  / *Condition*: required / *Type*: string /

  The path to destination file (either incl. file name or without file name)

- `bOverwrite`

  / *Condition*: optional / *Type*: bool / *Default*: False /

  - In case of the destination file already exists and `bOverwrite` is `True`, than the destination file will be overwritten.
  - In case of the destination file already exists and `bOverwrite` is `False` (default), than the destination file will not be overwritten and `MoveTo` returns `bSuccess = False`.

Returns:

- `bSuccess`

  / *Type*: bool /

  Indicates if the computation was successful or not

- `sResult`

  / *Type*: str /

  Contains details about what happens during computation

Chapter 4

CFolder.py

4.1 Function: rm_dir_readonly

Calls `os.chmod` in case of `shutil.rmtree` (within `Delete()`) throws an exception (making files writable).

4.2 Class: CFolder

```
PythonExtensionsCollection.Folder.CFolder
```

The class `CFolder` provides a small set of folder functions with extended parametrization (like switches defining if a folder is allowed to be overwritten or not).

Most of the functions at least returns `bSuccess` and `sResult`.

- `bSuccess` is `True` in case of no error occurred.
- `bSuccess` is `False` in case of an error occurred.
- `bSuccess` is `None` in case of a very fatal error occurred (exceptions).
- `sResult` contains details about what happens during computation.

Every instance of CFolder handles one single folder only and forces exclusive access to this folder.

It is not possible to create an instance of this class with a folder that is already in use by another instance.

The constructor of `CFolder` requires the input parameter `sFolder`, that is the path and the name of a folder that is handled by the current class instance.

4.2.1 Method: Delete

Deletes the current folder `sFolder`.

Arguments:

- `bConfirmDelete`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  Defines if it will be handled as error if the folder does not exist.
  If `True`: If the folder does not exist, the method indicates an error (`bSuccess = False`).
  If `False`: It doesn't matter if the folder exists or not.

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

4.2.2 Method: Create

Creates the current folder `sFolder`.

Arguments:

- `bOverwrite`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  - In case of the folder already exists and `bOverwrite` is `True`, than the folder will be deleted before creation.
  - In case of the folder already exists and `bOverwrite` is `False` (default), than the folder will not be touched.

  In both cases the return value `bSuccess` is `True` - because the folder exists.
- `bRecursive`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  - In case of `bRecursive` is `True`, than the complete destination path will be created (including all intermediate subfolders).
  - In case of `bRecursive` is `False`, than it is expected that the parent folder of the new folder already exists.

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

Chapter 5

CString.py

5.1 Class: CString

```
PythonExtensionsCollection.String.CString
```

The class `CString` contains some string computation methods like e.g. normalizing a path.

5.1.1 Method: NormalizePath

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 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`)

5.1.2 Method: DetectParentPath

Computes the path to any parent folder inside a given path. Optionally DetectParentPath is able to search for files inside the parent folder.

Arguments:

- `sStartPath`

  / *Condition*: required / *Type*: str /

  The path in which to search for a parent folder

- `sFolderName`

  / *Condition*: required / *Type*: str /

  The name of the folder to search for within `sStartPath`. It is possible to provide more than one folder name separated by semicolon

- `sFileName`

  / *Condition*: optional / *Type*: str / *Default*: None /

  The name of a file to search within the detected parent folder

Returns:

- `sDestPath`

  / *Type*: str /

  Path and name of parent folder found inside `sStartPath`, `None` in case of `sFolderName` is not found inside `sStartPath`. In case of more than one parent folder is found `sDestPath` contains the first result and `listDestPaths` contains all results.

- `listDestPaths`

  / *Type*: list /

  If `sFolderName` contains a single folder name this list contains only one element that is `sDestPath`. In case of `FolderName` contains a semicolon separated list of several folder names this list contains all found paths of the given folder names. `listDestPaths` is `None` (and not an empty list!) in case of `sFolderName` is not found inside `sStartPath`.

- `sDestFile`

  / *Type*: str /

  Path and name of `sFileName`, in case of `sFileName` is given and found inside `listDestPaths`. In case of more than one file is found `sDestFile` contains the first result and `listDestFiles` contains all results. `sDestFile` is `None` in case of `sFileName` is `None` and also in case of `sFileName` is not found inside `listDestPaths` (and therefore also in case of `sFolderName` is not found inside `sStartPath`).

- `listDestFiles`

  / *Type*: list /

  Contains all positions of `sFileName` found inside `listDestPaths`.

  `listDestFiles` is `None` (and not an empty list!) in case of `sFileName` is `None` and also in case of `sFileName` is not found inside `listDestPaths` (and therefore also in case of `sFolderName` is not found inside `sStartPath`).

- `sDestPathParent`

  / *Type*: str /

  The parent folder of `sDestPath`, `None` in case of `sFolderName` is not found inside `sStartPath` (`sDestPath` is `None`).

5.1.3 Method: StringFilter

During the computation of strings there might occur the need to get to know if this string fulfils certain criteria or not. Such a criterion can e.g. be that the string contains a certain substring. Also an inverse logic might be required: In this case the criterion is that the string does **not** contain this substring.

It might also be required to combine several criteria to a final conclusion if in total the criterion for a string is fulfilled or not. For example: The string must start with the string *prefix* and must also contain either the string *substring1* or the string *substring2* but must also **not** end with the string *suffix*.

This method provides a bunch of predefined filters that can be used singly or combined to come to a final conclusion if the string fulfils all criteria or not.

The filters are divided into three different types:

1. Filters that are interpreted as raw strings (called 'standard filters'; no wild cards supported)
2. Filters that are interpreted as regular expressions (called 'regular expression based filters'; the syntax of regular expressions has to be considered)
3. Boolean switches (e.g. indicating if also an empty string is accepted or not)

The input string might contain leading and trailing blanks and tabs. This kind of horizontal space is removed from the input string before the standard filters start their work (except the regular expression based filters).

The regular expression based filters consider the original input string (including the leading and trailing space).

The outcome is that in case of the leading and trailing space shall be part of the criterion, the regular expression based filters can be used only.

It is possible to decide if the standard filters shall work case sensitive or not. This decision has no effect on the regular expression based filters.

The regular expression based filters always work with the original input string that is not modified in any way.

Except the regular expression based filters it is possible to provide more than one string for every standard filter (must be a semikolon separated list in this case). A semicolon that shall be part of the search string, has to be masked in this way: `\;`.

This method returns a boolean value that is `True` in case of all criteria are fulfilled, and `False` in case of some or all of them are not fulfilled.

The default value for all filters is `None` (except `bSkipBlankStrings`). In case of a filter value is `None` this filter has no influence on the result.

In case of all filters are `None` (default) the return value is `True` (except the string itself is `None` or the string is empty and `bSkipBlankStrings` is `True`).

In case of the string is `None`, the return value is `False`, because nothing concrete can be done with `None` strings.

Internally every filter has his own individual acknowledge that indicates if the criterion of this filter is fulfilled or not.

The meaning of *criterion fulfilled* of a filter is that the filter supports the final return value `bAck` of this method with `True`.

The final return value `bAck` of this method is a logical join (`AND`) of all individual acknowledges (except `bSkipBlankStrings` and `sComment`; in case of their criteria are **not** fulfilled, immediately `False` is returned).

Summarized:

- Filters are used to define *criteria*
- The return value of this method provides the *conclusion* - indicating if all criteria are fulfilled or not

The following filters are available:

bSkipBlankStrings

- Like already mentioned above leading and trailing spaces are removed from the input string at the beginning
- In case of the result is an empty string and `bSkipBlankStrings` is `True`, the method immediately returns `False` and all other filters are ignored

sComment

- In case of the input string starts with the string `sComment`, the method immediately returns `False` and all other filters are ignored
- Leading blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- The idea behind this decision is: Ignore a string that is commented out

sStartsWith

- The criterion of this filter is fulfilled in case of the input string starts with the string `sStartsWith`
- Leading blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- More than one string can be provided (semicolon separated; logical join: `OR`)

sEndsWith

- The criterion of this filter is fulfilled in case of the input string ends with the string `sEndsWith`
- Trailing blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- More than one string can be provided (semicolon separated; logical join: `OR`)

sStartsNotWith

- The criterion of this filter is fulfilled in case of the input string does **not** start with the string `sStartsNotWith`
- Leading blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`

- More than one string can be provided (semicolon separated; logical join: `AND`)

sEndsNotWith

- The criterion of this filter is fulfilled in case of the input string does **not** end with the string `sEndsNotWith`
- Trailing blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- More than one string can be provided (semicolon separated; logical join: `AND`)

sContains

- The criterion of this filter is fulfilled in case of the input string contains the string `sContains` at any position
- Leading and trailing blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- More than one string can be provided (semicolon separated; logical join: `OR`)

sContainsNot

- The criterion of this filter is fulfilled in case of the input string does **not** contain the string `sContainsNot` at any position
- Leading and trailing blanks within the input string have no effect
- The decision also depends on `bCaseSensitive`
- More than one string can be provided (semicolon separated; logical join: `AND`)

sInclRegEx

- *Include* filter based on regular expressions (consider the syntax of regular expressions!)
- The criterion of this filter is fulfilled in case of the regular expression `sInclRegEx` matches the input string
- Leading and trailing blanks within the input string are considered
- `bCaseSensitive` has no effect
- A semicolon separated list of several regular expressions is **not** supported

sExclRegEx

- *Exclude* filter based on regular expressions (consider the syntax of regular expressions!)
- The criterion of this filter is fulfilled in case of the regular expression `sExclRegEx` does **not** match the input string
- Leading and trailing blanks within the input string are considered
- `bCaseSensitive` has no effect
- A semicolon separated list of several regular expressions is **not** supported

Further arguments:

- `sString`
  / *Condition*: required / *Type*: str /
  The input string that has to be investigated.

- `bCaseSensitive`

  / *Condition*: optional / *Type*: bool / *Default*: True /

  If `True`, the standard filters work case sensitive, otherwise not.

- `bDebug`

  / *Condition*: optional / *Type*: bool / *Default*: False /

  If `True`, additional output is printed to console (e.g. the decision of every single filter), otherwise not.

Returns:

- `bAck`

  / *Type*: bool /

  Final statement about the input string `sString` after filter computation

Examples:

1. Returns `True`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = "Sp",
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = "beats",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

2. Returns `False`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = "Sp",
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = "minute",
             sContains         = "beats",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

3. Returns `True`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
```

```
              sStartsWith       = None,
              sEndsWith         = None,
              sStartsNotWith    = None,
              sEndsNotWith      = None,
              sContains         = None,
              sContainsNot      = "Beats",
              sInclRegEx        = None,
              sExclRegEx        = None)
```

4. Returns `True`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = None,
             sContainsNot      = None,
             sInclRegEx        = r"\d{2}",
             sExclRegEx        = None)
```

5. Returns `False`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = "Speed",
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = None,
             sContainsNot      = None,
             sInclRegEx        = r"\d{3}",
             sExclRegEx        = None)
```

6. Returns `False`:

```
StringFilter(sString           = "Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = "Speed",
             sEndsWith         = "minute",
             sStartsNotWith    = "speed",
             sEndsNotWith      = None,
             sContains         = "beats",
             sContainsNot      = None,
             sInclRegEx        = r"\d{2}",
             sExclRegEx        = r"\d{2}")
```

7. Returns `False`:

```
StringFilter(sString           = "     ",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = None,
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

8. Returns `False`:

```
StringFilter(sString           = "# Speed is 25 beats per minute",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = "#",
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = "beats",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

9. Returns `False`:

```
StringFilter(sString           = "   Alpha is not beta; and beta is not
↪  gamma  ",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = "   Alpha ",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

Because blanks around search strings (here `"   Alpha "`) are considered, whereas the blanks around the input string are removed before computation. Therefore `"   Alpha "` cannot be found within the (shortened) input string.

10. This alternative solution returns `True`:

```
StringFilter(sString           = "   Alpha is not beta; and beta is not
↪  gamma  ",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
```

```
            sStartsWith       = None,
            sEndsWith         = None,
            sStartsNotWith    = None,
            sEndsNotWith      = None,
            sContains         = None,
            sContainsNot      = None,
            sInclRegEx        = r"\s{3}Alpha",
            sExclRegEx        = None)
```

11. Returns `True`:

```
StringFilter(sString           = "Alpha is not beta; and beta is not gamma",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = "beta; and",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

The meaning of `"beta; and"` is: The criterion is fulfilled in case of either `"beta"` or `" and"` can be found. That's `True` in this example - but this has nothing to do with the fact, that also this string `"beta; and"` can be found. Here the semikolon is a separator character and therefore part of the syntax.

A semicolon that shall be part of the search string, has to be masked with '`\;`'!

The meaning of `"beta\; not"` in the following example is: The criterion is fulfilled in case of `"beta; not"` can be found.

That's **not** `True`. Therefore the method returns `False`:

```
StringFilter(sString           = "Alpha is not beta; and beta is not gamma",
             bCaseSensitive    = True,
             bSkipBlankStrings = True,
             sComment          = None,
             sStartsWith       = None,
             sEndsWith         = None,
             sStartsNotWith    = None,
             sEndsNotWith      = None,
             sContains         = r"beta\; not",
             sContainsNot      = None,
             sInclRegEx        = None,
             sExclRegEx        = None)
```

5.1.4 Method: FormatResult

Formats the result string `sResult` depending on `bSuccess`:

- `bSuccess` is `True` indicates *success*
- `bSuccess` is `False` indicates an *error*
- `bSuccess` is `None` indicates an *exception*

Additionally the name of the method that causes the result, can be provided (*optional*). This is useful for debugging.

Arguments:

- `sMethod`
  / *Condition*: optional / *Type*: str / *Default*: (empty string) /
  Name of the method that causes the result.
- `bSuccess`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  Indicates if the computation of the method `sMethod` was successful or not.
- `sResult`
  / *Condition*: optional / *Type*: str / *Default*: (empty string) /
  The result of the computation of the method `sMethod`.

Returns:

- `sResult`
  / *Type*: str /
  The formatted result string.

Chapter 6

CUtils.py

6.1 Function: PrettyPrint

Wrapper function to create and use a `CTypePrint` object. This wrapper function is responsible for printing out the content to console and to a file (depending on input parameter).

The content itself is prepared by the method `TypePrint` of class `CTypePrint`. This happens `PrettyPrint` internally.

The idea behind the `PrettyPrint` function is to resolve also the content of composite data types and provide for every parameter inside:

- 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

Example call:

```
PrettyPrint(oData)
```

(*with oData is a Python variable of any type*)

The output can e.g. look like this:

```
[DICT] (3/1) > {K1} [STR]  :  'Val1'
[DICT] (3/2) > {K2} [LIST] (4/1) > [INT]  :  1
[DICT] (3/2) > {K2} [LIST] (4/2) > [STR]  :  'A'
[DICT] (3/2) > {K2} [LIST] (4/3) > [INT]  :  2
[DICT] (3/2) > {K2} [LIST] (4/4) > [TUPLE] (2/1) > [INT]  :  9
[DICT] (3/2) > {K2} [LIST] (4/4) > [TUPLE] (2/2) > [STR]  :  'Z'
[DICT] (3/3) > {K3} [INT]  :  5
```

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

For example the meaning of the fifth line of output

```
[DICT] (3/2) > {K2} [LIST] (4/4) > [TUPLE] (2/1) > [INT]  :  9
```

is:

- The type of input parameter (`oData`) is `dict`

- The dictionary contains 3 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 `tuple`
- The tuple contains 2 elements
- The current line gives information about the first element of the tuple
- The first element of the tuple is of type `int` and has the value 9

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

Arguments:

- `oData`
  / *Condition*: required / *Type*: (*any Python data type*) /
  A variable of any Python data type.
- `hOutputFile`
  / *Condition*: optional / *Type*: file handle / *Default*: None /
  If handle is not `None` the content is written to this file, otherwise not.
- `bToConsole`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  If `True` the content is written to console, otherwise not.
- `nIndent`
  / *Condition*: optional / *Type*: int / *Default*: 0 /
  Sets the number of additional blanks at the beginning of every line of output (indentation).
- `sPrefix`
  / *Condition*: optional / *Type*: str / *Default*: None /
  Sets a prefix string that is added at the beginning of every line of output.
- `bHexFormat`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `True` the output is printed in hexadecimal format (but valid for strings only).

Returns:

- `listOutLines` (*list*)
  / *Type*: list /
  List of lines containing the prepared output

6.2 Class: CTypePrint

```
PythonExtensionsCollection.Utils.CUtils
```

The class `CTypePrint` provides a method (`TypePrint`) to compute the following data:

- 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

of simple and composite data types.
The call of this method is encapsulated within the function `PrettyPrint` inside this module.

6.2.1 Method: TypePrint

The method `TypePrint` computes details about the input variable `oData`.
Arguments:

- `oData`
  / *Condition*: required / *Type*: any Python data type /
  Python variable of any data type.
- `bHexFormat`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `True` the output is provide in hexadecimal format.

Returns:

- `listOutLines`
  / *Type*: list /
  List of lines containing the resolved content of `oData`.

Chapter 7

Appendix

About this package:

Table 7.1: Package setup

Setup parameter	Value
Name	PythonExtensionsCollection
Version	0.7.1
Date	14.06.2022
Description	Additional Python functions
Package URL	python-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 8

History

0.1.0	08/2021
Initial version	
0.2.0	02/2022
Code maintenance	
0.3.0	20.05.2022
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	10.06.2022
Module `CFolder` *added (with methods:* `Create` *and* `Delete`	

PythonExtensionsCollection.pdf
Created at 14.06.2022 - 17:49:25
by GenPackageDoc v. 0.16.0

4.2 RobotframeworkExtensions

RobotframeworkExtensions

v. 0.6.0

Holger Queckenstedt

02.06.2022

Contents

A

Chapter 1

Introduction

The *Robotframework Extensions Collection* extends the functionality of the Robotframework by some keywords providing features, that are implemented in the *Python Extensions Collection*.

The goal behind these extensions is to have certain functionality available in both: pure Python applications and Robotframework.

The *Robotframework Extensions Collection* requires an installed *Python Extensions Collection*, that can be found in this repository:

python-extensions-collection

Chapter 2

Collection.py

The Collection module is the interface between the Python Extensions Collection and the Robotframework.
This library containing the keyword definitions, can be imported in the following way:

```
Library    RobotframeworkExtensions.Collection    WITH NAME    rf.extensions
```

2.1 Class: Collection

```
RobotframeworkExtensions.Collection
```

Module main class

2.1.1 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 logging.
The output contains for every parameter: the value, the type and counter values (in case of composite data types).
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).

Example:
Variable of Python type `list`:

```
set_test_variable    @{aItems}    String
...                                ${25}
...                                ${True}
...                                ${None}
```

Call of `pretty_print` keyword:

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

Output:

```
INFO - [LIST] (4/1) > [STR]  :  'String'
INFO - [LIST] (4/2) > [INT]  :  25
INFO - [LIST] (4/3) > [BOOL]  :  True
INFO - [LIST] (4/4) > [NONE]  :  None
```

2.1.2 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`)

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:

```
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

Appendix

About this package:

Table 3.1: Package setup

Setup parameter	Value
Name	RobotframeworkExtensions
Version	0.6.0
Date	02.06.2022
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 4

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*	

RobotframeworkExtensions.pdf

Created at 14.06.2022 - 17:50:00

by GenPackageDoc v. 0.16.0

4.3 GenPackageDoc

GenPackageDoc

v. 0.16.0

Holger Queckenstedt

01.06.2022

Contents

A

Chapter 1

Introduction

The Python package `GenPackageDoc` generates the documentation of Python modules. The content of this documentation is taken out of the docstrings of functions, classes and their methods.

It is possible to extend the documentation by the content of additional text files. The docstrings and also the additional text files have to be written in rst syntax (rst is the abbreviation for "**r**e **s**tructured **t**ext", that is a certain markdown dialect).

The documentation is generated in two steps:

1. The rst sources are converted into LaTeX sources
2. The LaTeX sources are converted into a PDF document. This requires a separately installed LaTeX distribution (recommended: MiKTeX), that is **not** part of `GenPackageDoc`.

The sources of `GenPackageDoc` are available in the following GitHub repository:

python-genpackagedoc

The repository `python-genpackagedoc` uses it's own functionality to document itself and the contained Python package `GenPackageDoc`.

Therefore the complete repository can be used as an example about writing a package documentation.

It has to be considered, that the main goal of `GenPackageDoc` is to document Python sources that are stored within a repository, and therefore we have dependencies to the structure of the repository. For example: Configuration files with values that are specific for a repository, should not be installed. Such a specific configuration value is e.g. the name of the package or the name of the PDF document.

The impact is: There is a deep relationship between the repository containing the sources to be documented, and the sources and the configuration of `GenPackageDoc` itself. Therefore some manual preparations are necessary to use `GenPackageDoc` also in other repositories.

How to do this is explained in detail in the next chapters.

The outcome of all preparations of `GenPackageDoc` in your own repository is a PDF document like the one you are currently reading.

Chapter 2

Description

2.1 Repository content

What is the content of the repository `python-genpackagedoc`*?*

- Folder `GenPackageDoc`

  Contains the package code.

  This folder is specific for the package.

- Folder `config`

  Contains the repository configuration (e.g. the name of the package, the name of the repository, the author, and more ...).

  This folder is specific for the repository.

- Folder `additions`

  Contains additionally needed sources like setup related class definitions and sources, that are imported from other repositories - to make this repository stand alone

- Folder `packagedoc`

  Contains all package documentation related files, e.g. the `GenPackageDoc` configuration, additional input files and the generated documentation itself.

  This folder is specific for the documentation.

- Repository root folder

  - `genpackagedoc.py`
    Python script to start the documentation build
  - `setup.py`
    Python script to install the package sources. This includes the execution of `genpackagedoc.py`. Therefore building the documentation is part of the installation process.
  - `dump_repository_config.py`
    Little helper to dump the repository configuration to console

2.2 Documentation build process

How do the files and folders listed above, belong together? What is the way, the information flows when the documentation is generated?

- The process starts with the execution of `genpackagedoc.py` within the repository root folder.

  `genpackagedoc.py` can be used stand alone - but this script is also called by `setup.py`. The impact is that every installation includes an update of the documentation.

- `genpackagedoc.py` creates a repository configuration object

  `config/CRepositoryConfig.py`

- The repository configuration object reads the static repository configuration values out of a separate json file

  `config/repository_config.json`

- The repository configuration object adds dynamic values (like operating system specific settings and paths) to the repository configuration. Not all of them are required for the documentation build process, but the repository configuration also supports the setup process (`setup.py`).

  There is one certain setting in the repository configuration file

  `config/repository_config.json`,

  that is essential for the documentation build process:

  `"PACKAGEDOC" : "./packagedoc"`

  This is the path to a folder, in which all further documentation related files are placed. In case of the path is relative, the reference is the position of `genpackagedoc.py`. It is required that within this folder the configuration file for the documentation build process

  `packagedoc_config.json`

  can be found. The name of this json file is fix!

- The configuration file `packagedoc_config.json` contains settings like

  - Paths to Python packages to be documented
  - Paths and names of additional rst files
  - Path and name of output folder (tex files and output PDF file)
  - User defined parameter (that can be defined here as global runtime variables and can be used in any rst code)
  - Basic settings related to the output PDF file (like document name, name of author, ...)
  - Path to LaTeX compiler
    (*a LaTeX distribution is not part of* `GenPackageDoc`)

  Be aware of that the within `packagedoc_config.json` specified output folder

  `"OUTPUT" : "./build"`

  will be deleted at the beginning of the documentation build process! Make sure that you do not have any files inside this folder opened when you start the process. In case of the path is relative, the reference is the position of `genpackagedoc.py`. The complete path is created recursively.

  Further details are explained within the json file itself.

- `genpackagedoc.py` also creates an own configuration object

GenPackageDoc/CPackageDocConfig.py

CPackageDocConfig.py takes over all repository configuration values, reads in the static GenPackageDoc configuration (packagedoc_config.json) and adds dynamically computed values like the full absolute paths belonging to the documentation build process. Also all command line parameters are resolved and checked.

The reference for all relative paths is the position of genpackagedoc.py (that is the repository root folder).

After the execution of genpackagedoc.py the resulting PDF document can be found under the specified name within the specified output folder ("OUTPUT"). This folder also contains all temporary files generated during the documentation build process.

Because the output folder is a temporary one, the PDF document is copied to the folder containing the package sources and therefore is included in the package installation. This is defined in the GenPackageDoc configuration, section "PDFDEST".

Command line

Some configuration parameter predefined within packagedoc_config.json, can be overwritten in command line.

--output

Path and name of folder containing all output files.

--pdfdest

Path and name of folder in which the generated PDF file will be copied to (after this file has been created within the output folder).

Caution: The generated PDF file will per default be copied to the package folder within the repository. This is defined in packagedoc_config.json. The version of the PDF file within the package folder will be part of the installation (when using setup.py). When you change the PDF destination, then you get this file at another location - but this file will not be part of the installation any more. Installed will be the version, that is still present within the package folder of the repository. Please try to get the bottom of your motivation when you change this setting.

--strict

If True, a missing LaTeX compiler aborts the process, otherwise the process continues.

Example

```
genpackagedoc.py --output="../any/other/location" --pdfdest="../any/other"
 ↪  --strict=true
```

All listed parameters are optional. GenPackageDoc creates the complete output path (recursively). The PDF destinatiopn path is expected to be existing already.

2.3 PDF document structure

How is the resulting PDF document structured? What causes an entry within the table of content of the PDF document?

In the following we use terms taken over from the LaTeX world: *chapter*, *section* and *subsection*.

A *chapter* is the top level within the PDF document; a *section* is the level below *chapter*, a *subsection* is the level below *section*.

The following assignments happen during the generation of a PDF document:

- The content of every additionally included separate rst file is a *chapter*.
  - In case of you want to add another chapter to your documentation, you have to include another rst file.
  - The headline of the chapter is the name of the rst file (automatically).
    Therefore the heading within an rst file has to start at section level!
- The content of every included Python module is also a *chapter*.
  - The headline of the chapter is the name of the Python module (automatically).
    This means also that within the PDF document structure every Python module is at the same level as additionally included rst files.
- Within additionally included separate rst files sections and subsections can be defined by the following rst syntax elements for headings:
  - A line underlined with "=" characters is a section
  - A line underlined with "–" characters is a subsection
- Within the docstrings of Python modules the headings are added automatically (for functions, classes and methods)
  - Classes and functions are listed at section level (both classes and functions are assumed to be at the same level).
  - Class methods are listed at subesction level.

  Further nestings of headings are not supported (because we do not want to overload the table of content).

2.4 Examples

2.4.1 Example 1: rst file

The text of this chapter is taken over from an rst file named `Description.rst`.

This rst file contains the following headlines:

```
Repository content
==================

Documentation build process
===========================

PDF document structure
======================

Examples
========

Example 1: rst file
-------------------

Example 2: Python module
------------------------
```

Because `Description.rst` is the second imported rst file, the chapter number is 2. The chapter headline is "Description" (the name of the rst file). The top level headlines *within* the rst file are at *section* level. The fourth section (Examples) contains two subsections.

The outcome is the following part of the table of content:

2.4.2 Example 2: Python module

Part of this documentation is a Python module with name `CDocBuilder.py` (listed in table of content at *chapter* level). This module contains a class with name `CDocBuilder` (listed in table of content at *section* level). The class `CDocBuilder` contains a method with name `Build` (listed in table of content at *subsection* level).

This causes the following entry within the table of contents:

2.5 Interface and module descriptions

How to describe an interface of a function or a method? How to describe a Python module?

To have a unique look and feel of all interface descriptions, the following style is recommended:

Example

```
      """
Description of function or method.

**Arguments:**

* ``input_param_1``

  / *Condition*: required / *Type*: str /

  Description of input_param_1.

* ``input_param_2``

  / *Condition*: optional / *Type*: bool / *Default*: False /

  Description of input_param_2.

**Returns:**

* ``return_param``

  / *Type*: str /

  Description of return_param.
      """
```

Some of the special characters used within the interface description, are part of the rst syntax. They will be explained in one of the next sections.

The docstrings containing the description, have to be placed directly in the next line after the `def` or `class` statement.

It is also possible to place a docstring at the top of a Python module. The exact position doesn't matter - but it has to be the first constant expression within the code. Within the documentation the content of this docstring is placed before the interface description and should contain general information belonging to the entire module.

The usage of such a docstring is an option.

2.6 Runtime variables

What are "runtime variables" and how to use them in rst text?

All configuration parameters of `GenPackageDoc` are taken out of four sources:

1. the static repository configuration
   `config/repository_config.json`
2. the dynamic repository configuration
   `config/CRepositoryConfig.py`
3. the static `GenPackageDoc` configuration
   `packagedoc/packagedoc_config.json`
4. the dynamic `GenPackageDoc` configuration
   `GenPackageDoc/CPackageDocConfig.py`

Some of them are runtime variables and can be accessed within rst text (within docstrings of Python modules and also within separate rst files).

This means it is possible to add configuration values automatically to the documentation.

This happens by encapsulating the runtime variable name in triple hashes. This "triple hash" syntax is introduced to make it easier to distinguish between the json syntax (mostly based on curly brackets) and additional syntax elements used within values of json keys.

The name of the repository e.g. can be added to the documentation with the following rst text:

```
The name of the repository is ###REPOSITORYNAME###.
```

This document contains a chapter "Appendix" at the end. This chapter is used to make the repository configuration a part of this documentation and can be used as example.

Additionally to the predefined runtime variables a user can add own ones.
See `"PARAMS"` within `packagedoc_config.json`.

All predefined runtime variables are written in capital letters. To make it easier for a developer to distinguish between predefined and user defined runtime variables, all user defined runtime variables have to be written in small letters completely.

Also the `"DOCUMENT"` keys within `packagedoc_config.json` are runtime variables.

Also within `packagedoc_config.json` the triple hash syntax can be used to access repository configuration values.

With this mechanism it is e.g. possible to give the output PDF document automatically the name of the package:

```
"DOCUMENT" : {
                "OUTPUTFILENAME" : "###PACKAGENAME###.tex",
```

2.7 Syntax aspects

Important to know about the syntax of Python and rst is:

- In both Python and rst the indentation of text is part of the syntax!
- The indentation of the triple quotes indicating the beginning and the end of a docstring has to follow the Python syntax rules.
- The indentation of the content of the docstring (= the interface description in rst format) has to follow the rst syntax rules. To avoid a needless indentation of the text within the resulting PDF document it is recommended to start the docstring text within the first column (or rather use the first column as reference for further indentations of rst text).
- In rst also blank lines are part of the syntax!

Please be attentive while typing your documentation in rst format!

2.7.1 Syntax extensions

`GenPackageDoc` extends the rst syntax by the following topics:

- *newline*

  A newline (line break) is realized by a slash ('/') at the end of a line containing any other rst text (this means: the slash must **not** be the only character in line).
  Internally this slash is mapped to the LaTeX command `\newline`.

- *vspace*

  An additional vertical space (size: the height of the 'x' character - depending on the current type and size of font) is realized by a single slash ('/'). This slash must be the only character in line!
  Internally this slash is mapped to the LaTeX command `\vspace{1ex}`.

- *newpage*

  A newpage (page break) is realized by a double slash ('//'). These two slashes must be the only characters in line!
  Internally this double slash is mapped to the LaTeX command `\newpage`.

These syntax extensions can currently be used in separate rst files only and are not available within docstrings of Python modules.

2.7.2 Examples

(to be continued)

Chapter 3

CDocBuilder.py

Python module containing all methods to generate tex sources.

3.1 Class: CDocBuilder

```
GenPackageDoc.CDocBuilder
```

Main class to build tex sources out of docstrings of Python modules and separate text files in rst format.

Depends on a json configuration file, provided by a `oPackageDocConfig` object (this includes the Repository configuration).

Method to execute: `Build()`

3.1.1 Method: Build

Arguments:

(*no arguments*)

Returns:

- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method `sMethod` was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method `sMethod`.

Chapter 4

CPackageDocConfig.py

Python module containing the configuration for GenPackageDoc. This includes the repository configuration and command line values.

4.1 Class: CPackageDocConfig

```
GenPackageDoc.CPackageDocConfig
```

4.1.1 Method: PrintConfig

Prints all cofiguration values to console.

4.1.2 Method: PrintConfigKeys

Prints all cofiguration key names to console.

4.1.3 Method: Get

Returns the configuration value belonging to a key name.

4.1.4 Method: GetConfig

Returns the complete configuration dictionary.

Chapter 5

CPatterns.py

Python module containing source patterns used to generate the tex file output.

5.1 Class: CPatterns

```
GenPackageDoc.CPatterns
```

The `CPatterns` class provides a set of LaTeX source patterns used to generate the tex file output.

All source patterns are accessible by corresponding `Get` methods. Some source patterns contain placeholder that will be replaced by input parameter of the `Get` method.

5.1.1 Method: GetHeader

Defines the header of the main tex file.

Arguments:

- `sTitle`
  / *Condition*: required / *Type*: str /
  The title of the output document (name of the described package)
- `sVersion`
  / *Condition*: required / *Type*: str /
  The version of the output document (version of the described package)
- `sAuthor`
  / *Condition*: required / *Type*: str /
  The author of the output document (author of the described package)
- `sDate`
  / *Condition*: required / *Type*: str /
  The date of the output document (date of the described package)

Returns:

- `sHeader`
  / *Type*: str /
  LaTeX code containing the header of main tex file.

5.1.2 Method: GetChapter

Defines single chapter of the main tex file.

A single chapter is equivalent to an additionally imported text file in rst format or equivalent to a single Python module within a Python package.

Arguments:

- `sHeadline`

  / *Condition*: required / *Type*: str /

  The chapter headline (that is either the name of an additional rst file or the name of a Python module).

- `sDocumentName`

  / *Condition*: required / *Type*: str /

  The name of a single tex file containing the chapter content. This file is imported in the main text file after the chapter headline that is set by `sHeadline`.

Returns:

- `sHeader`

  / *Type*: str /

  LaTeX code containing the headline and the input of a single tex file.

5.1.3 Method: GetFooter

Defines the footer of the main tex file.

Arguments:

(*no arguments*)

Returns:

- `sFooter`

  / *Type*: str /

  LaTeX code containing the footer of the main tex file.

Chapter 6

CSourceParser.py

Python module containing all methods to parse the documentation content of Python source files.

6.1 Class: CSourceParser

```
GenPackageDoc.CSourceParser
```

The `CSourceParser` class provides a method to parse the functions, classes and their methods together with the corresponding docstrings out of Python modules. The docstrings have to be written in rst syntax.

6.1.1 Method: ParseSourceFile

The method `ParseSourceFile` parses the content of a Python module.

Arguments:

- `sFile`
  / *Condition*: required / *Type*: str /
  Path and name of a single Python module.
- `bIncludePrivate`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `False`: private methods are skipped, otherwise they are included in documentation.
- `bIncludeUndocumented`
  / *Condition*: optional / *Type*: bool / *Default*: True /
  If `True`: also classes and methods without docstring are listed in the documentation (together with a hint that information is not available), otherwise they are skipped.

Returns:

- `dictContent`
  / *Type*: dict /
  A dictionary containing all the information parsed out of `sFile`.
- `bSuccess`
  / *Type*: bool /
  Indicates if the computation of the method `sMethod` was successful or not.

- `sResult`
  / *Type*: str /
  The result of the computation of the method `sMethod`.

Chapter 7

Outlook

ToDo list:

- [01]

  Introduce `setup.py` including the execution of `genpackagedoc.py` and adding the generated PDF document to the installation.
  Introduce README.rst and README.md.

  10.05.2022: Setup process introduced and README.rst added

- [02]

  Currently it is hard coded, that private functions and methods are skipped. Therefore they are not part of the resulting PDF document.
  A configuration switch might be useful to give the user the ability to control this behavior.

  09.05.2022: Parameter 'INCLUDEPRIVATE' added

- [03]

  Currently it is implemented that also functions, classes and methods without docstrings are part of the resulting PDF document. They are listed together with the hint, that a docstring is not available.
  A configuration switch might be useful to give the user the ability to control this behavior.

  09.05.2022: Parameter 'INCLUDEUNDOCUMENTED' added

- [04]

  Currently it is implemented that for Python modules will be searched recursively within the given root folder. Maybe the algorithm also catches modules from which the user does not want `GenPackageDoc` to include them.
  A configuration exclude filter can be implemented to skip those files. Or maybe other way round: An include filter includes a subset of available files only.

  The same filter mechanism can be extended for the content of Python modules (= include/exclude functions, classes and methods).

- [05]

  Currently the configuration parameter for the documentation build proccess are taken from a json file `packagedoc_config.json`.
  It might be helpful to have the possibility to overwrite them in command line (e.g. for redirecting the path to the output folder without changing any code).

  31.05.2022: Implemented in v. 0.15.0

- [06]

  Introduce text boxes for warnings, errors and informations.

  19.05.2022: implemented in v. 0.12.0

- [07]

  The documentation build process allows relative paths only
  (in `packagedoc_config.json`).
  Maybe a mechanism is useful to allow absolute paths and paths based on environment variables also.

  01.06.2022: implemented in v. 0.16.0

- [08]

  Explore further rst syntax elements like the code directive. Some of them produces LaTeX code that requires the include of additional LaTeX packages. Sometimes this causes errors that have to be fixed.

  05.05.2022: Python syntax highlighting realized

- [09]

  The documentation has to be extended by a set of rst examples (rst best practices).

- [10]

  A postprocessing for LaTeX code needs to be implemented:

  - Enable proper line breaks
  - Resolve the ambiguity of labels created automatically when the LaTeX code is generated (for every input file separately)

  10.05.2022: Experimental syntax extensions for `newline`, `newpage` and `vspace`

  17.05.2022: Postprocessing for rst and tex sources added; "multiply-defined labels" fix.

- [11]

  Currently the docstrings of Python modules have to contain a heading for functions, classes and methods. The developer is responsible for that. Maybe it is not necessary to maintain these headings manually. It has to be investigated, if these headings can be added automatically by `GenPackageDoc`.

  06.05.2022: Headings are added automatically.

- [12]

  Currently the documentation of a single Python module starts at *function* or *class* level. This means it is not possible to provide common information about the Python module itself (placed **before** the content of the first function or class of the module). A way have to be found to add such content.

  06.05.2022: Implemented in version 0.4.0

- [13]

  The error handling needs to be extended!

- [14]

  Take over the description of

  ```
  config/repository_config.json
  ```

  from inside this json file (comment blocks) to the main PDF document.

- [15]

  Reference section with useful links

- [16]

  History

  10.05.2022: History added

- [17]

  Debug switch to enable additional output

- [18]

  Parse decorators to identify Robot Framework keyword definitions

- [19]

  Selftest

- [20]

  Introduce a separate folder containing TeX styles - instead of having them hard coded within CPatterns.py.

  24.05.2022: implemented in v. 0.13.0

Chapter 8

Appendix

About this package:

Table 8.1: Package setup

Setup parameter	Value
Name	GenPackageDoc
Version	0.16.0
Date	01.06.2022
Description	Documentation builder for Python packages
Package URL	python-genpackagedoc
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 :: 3 - Alpha
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 9

History

0.1.0	04/2022
Initial version	
0.2.0	05.05.2022
Python syntax highlighting within code blocks added	
0.3.0	06.05.2022
Automated headings for functions, classes and methods	
0.4.0	06.05.2022
Possibility to describe complete Python modules added	
0.5.0	09.05.2022
Parameter `INCLUDEPRIVATE` added	
0.6.0	09.05.2022
Parameter `INCLUDEUNDOCUMENTED` added	
0.7.0	10.05.2022
Setup process introduced and README.rst added; code maintenance	
0.8.0	10.05.2022
Bugfixes and code maintenance; history added	
0.9.0	10.05.2022
Layout maintenance and syntax extensions for `newline`, `newpage` and `vspace` reworked	
0.9.1	11.05.2022
Documentation maintenance	
0.9.2	16.05.2022
Fix: automated line breaks within code blocks	

0.10.0	17.05.2022
Postprocessing for rst and tex sources added; 'multiply-defined labels' fix.	
0.11.0	18.05.2022
Import of tex files enabled	
0.12.0	19.05.2022
Admonitions added, based on LaTeX environment 'tcolorbox'; *layout adaptions in titlepage;* *page numbering fix in TOC*	
0.13.0	24.05.2022
LaTeX style definitions moved to separate folder	
0.14.0	27.05.2022
LaTeX compiler check added; *control parameter* `STRICT` *added to* `packagedoc_config`	
0.15.0	31.05.2022
Command line added; *separate GenPackageDoc configuration class added*	
0.16.0	01.06.2022
Path computation reworked	

GenPackageDoc.pdf

Created at 14.06.2022 - 17:50:09

by GenPackageDoc v. 0.16.0