python-genpackagedoc

Holger Queckenstedt

11.04.2022

Contents

Chapter 1

Introduction

1.1 Introduction part 1

part 1 part 1

1.1.1 Introduction part 1 a

part 1 a part 1 a

1.1.2 Introduction part 1 b

part 1 b part 1 b

1.2 Introduction part 2

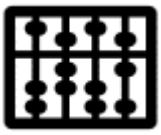
part 2 part 2

1.2.1 Introduction part 2 a

part 2 a part 2 a

1.2.2 Introduction part 2 b

part 2 b part 2 b



Chapter 2

Description

2.1 Description part 1

A	not A
False	True
True	False

A	not A
False	True
True	False

A	not A
False	True
True	False

2.2 Description part 2

A	not A
False	True
True	False

A	not A
False	True
True	False

A	not A
False	True
True	False

Chapter 3

CDummy.py

3.1 Function: DummyFunction

This is a dummy function implemented by Holger Queckenstedt as part of package GenPackageDoc.

The soures can be found here

> https://github.com/test-fullautomation/python-genpackagedoc.

Feel free to send a mail to Holger.Queckenstedt@de.bosch.com.

Please make sure to have installed Python with version >=3.0.

An additional hint is given 13.04.2022 - 10:58:41

> Please send a card to Karl Otto

3.2 Function: DummyFunctionNoDocString_1

docstring not available

3.3 Function: DummyFunctionNoDocString_2

docstring not available

3.4 Class: DummyClassNoDocString_1

Import: `PythonExtensionsCollection.Dummy.CDummy`

docstring not available

3.4.1 Method: DummyMethodNoDocString_1_1

docstring not available

3.4.2 Method: DummyMethodNoDocString_1_2

docstring not available

3.5 Class: DummyClassNoDocString_2

Import: `PythonExtensionsCollection.Dummy.CDummy`
docstring not available

3.5.1 Method: DummyMethodNoDocString_2_1

docstring not available

3.5.2 Method: DummyMethodNoDocString_2_2

docstring not available

Chapter 4

CFile.py

4.1 Class: enFileStatiType

Import: `PythonExtensionsCollection.File.CFile`

Enumeration class that defines values for file status.

4.2 Class: CFile

Import: `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.

4.2.1 Method: Close

Closes the opened file.

Arguments:

(no args)

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`.

4.2.2 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 `sMethod` was successful or not.

- `sResult`

  / *Type*: str /

  The result of the computation of the method `sMethod`.

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

4.2.4 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 `sMethod` was successful or not.

- `sResult`

  / *Type*: str /

  The result of the computation of the method `sMethod`.

4.2.5 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 / `bToScreen` | `Condition:` optional | `Type:` bool | `Default:` False |

  If `True`, the content read from file is also printed to screen, otherwise not.

4.2.6 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`

4.2.7 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 `sMethod` was successful or not.

- `sResult`

  / *Type*: str /

  The result of the computation of the method `sMethod`.

4.2.8 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 5

CString.py

5.1 Class: CString

Import: `PythonExtensionsCollection.String.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 compotation

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. A semicolon that shall be part of the search, has to be masked!

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

Import: `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

7.1 About this package

Package name:

GenPackageDoc

Package version

0.1.0

Package date

11.04.2022

Package description

This package provides a dcoumentation builder for Python packages

Package URL on GitHub

https://github.com/test-fullautomation/python-genpackagedoc

Author of package

Holger Queckenstedt

Email of author

Holger.Queckenstedt@de.bosch.com

Package programming language

Programming Language :: Python :: 3

Package license

License :: OSI Approved :: Apache Software License

Supported operating systems

Operating System :: OS Independent

Python required

>=3.0

Development status

Development Status :: 2 - Pre-Alpha

Intended audience

Intended Audience :: Developers

Package topic

Topic :: Software Development

This PDF has been created at 13.04.2022 - 10:58:41
