PythonExtensionsCollection

v. 0.11.5

Holger Queckenstedt

21.02.2023

Contents

Chapter 1

Introduction

The **PythonExtensionsCollection** extends the functionality of Python by some useful functions that are not available in Python immediately.

This covers for example file and folder operations, string operations like normalizing a path and a pretty print method.

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

The sources of the **PythonExtensionsCollection** are available in GitHub.

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

Chapter 2

Description

2.1 Modules

The **PythonExtensionsCollection** contains the following modules:

1. `CComparison`
   Compare two text files based on regular expressions.
   Import: `from PythonExtensionsCollection.Comparison.CComparison import CComparison`

2. `CFile`
   File operations like read, write, copy, move, ...
   Import: `from PythonExtensionsCollection.File.CFile import CFile`

3. `CFolder`
   Folder operations like create, delete, copy, ...
   Import: `from PythonExtensionsCollection.Folder.CFolder import CFolder`

4. `CString`
   String operations like normalize a path, string filter, format results, ...
   Import: `from PythonExtensionsCollection.String.CString import CString`

5. `CUtils`
   Pretty print of Python data types
   Import: `from PythonExtensionsCollection.Utils.CUtils import CTypePrint`

2.2 Methods

Additionally to the interface descriptions in the second part of this document this section contains a more detailed description of some assorted methods together with examples how to use.

2.2.1 String operations with CString

NormalizePath

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

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

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

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

Preparing a path for best usage within code also includes collapsing redundant separators and up-level references. Python already provides functions to do this, but the outcome (path contains slashes or backslashes) depends on the

operating system. And like already mentioned above also under Windows backslashes might not be the preferred choice.

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

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

Example 1 (*file system path*)

```
path = r"C:\\subfolder1///../subfolder2\\\\../subfolder3\\"
path = CString.NormalizePath(path)
print(path)
```

Result (*output contains slashes*)

```
C:/subfolder3
```

Example 2 (*file system path*)

```
path = r"C:\\subfolder1///../subfolder2\\\\../subfolder3\\"
path = CString.NormalizePath(path, bWin=True)
print(path)
```

Result (*output contains masked backslashes*)

```
C:\\subfolder3
```

Example 3 (*path of a local network resource*)

```
path = r"\\anyserver.com\\part1//part2\\\\part3/part4"
path = CString.NormalizePath(path)
print(path)
```

Result

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

Example 4 (*internet address*)

```
path = r"http:\\anyserver.com\\part1//part2\\\\part3/part4"
path = CString.NormalizePath(path)
print(path)
```

Result

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

DetectParentPath

The method `DetectParentPath` computes the path to any parent folder inside a given path. Optionally `DetectParentPath` is able to search for files inside the identified parent folder.

In the following examples we assume to have the following file system structure:

```
D:\pathtest\ABC\DEF\GHI\FILE.txt
D:\pathtest\ABC\FILE.txt
D:\pathtest\RST\UVW\XYZ\FILE.txt
```

We are inside folder `GHI` and want to know the path to folder `ABC` .

Python code

```
sStartPath  = r"D:\pathtest\ABC\DEF\GHI"
sFolderName = "ABC"
sDestPath, listDestPaths, sDestFile, listDestFiles, sDestPathParent = ↩
    ↪ CString.DetectParentPath(sStartPath, sFolderName)
print(f"sDestPath       = {sDestPath}")
print(f"listDestPaths   = {listDestPaths}")
print(f"sDestFile       = {sDestFile}")
print(f"listDestFiles   = {listDestFiles}")
print(f"sDestPathParent = {sDestPathParent}")
```

Outcome

```
sDestPath       = D:/pathtest/ABC
listDestPaths   = ['D:/pathtest/ABC']
sDestFile       = None
listDestFiles   = None
sDestPathParent = D:/pathtest
```

`sDestPath` contains the path to `sFolderName` within `sStartPath` . `sDestPathParent` contains the path to the parent folder of `sDestPath` . The meaning of the remaining return parameter are handled in the next examples.

It is possible to search for several folders.

We are inside folder `GHI` and want to know the path to folders `ABC` and `DEF` .

```
sStartPath  = r"D:\pathtest\ABC\DEF\GHI"
sFolderName = "ABC;DEF"
sDestPath, listDestPaths, sDestFile, listDestFiles, sDestPathParent = ↩
    ↪ CString.DetectParentPath(sStartPath, sFolderName)
```

Outcome

```
sDestPath       = D:/pathtest/ABC/DEF
listDestPaths   = ['D:/pathtest/ABC/DEF', 'D:/pathtest/ABC']
sDestFile       = None
listDestFiles   = None
sDestPathParent = D:/pathtest/ABC
```

`sFolderName` is a semicolon separated list of folder names. Accordingly to this list of folder names `listDestPaths` contains the paths to these folders. `DetectParentPath` searches in the path from right to left (from bottom level up to top level). Therefore the folder `DEF` is the first one who is found. The order of elements in `listDestPaths` is not synchronized with the order of folder names in `sFolderName` ! In every case `sDestPath` contains the first element in the list. `sDestPathParent` contains the path to the parent folder of `sDestPath` .

It is possible to search for a file.

We are inside folder `GHI` and want to know the path to folder `DEF` and within this folder we want to know the path to file `FILE.txt` .

```
sStartPath  = r"D:\pathtest\ABC\DEF\GHI"
sFolderName = "DEF"
sFileName   = "FILE.txt"
sDestPath, listDestPaths, sDestFile, listDestFiles, sDestPathParent = ↩
    ↪ CString.DetectParentPath(sStartPath, sFolderName, sFileName)
```

Outcome

```
sDestPath       = D:/pathtest/ABC/DEF
listDestPaths   = ['D:/pathtest/ABC/DEF']
sDestFile       = D:/pathtest/ABC/DEF/GHI/FILE.txt
listDestFiles   = ['D:/pathtest/ABC/DEF/GHI/FILE.txt']
sDestPathParent = D:/pathtest/ABC
```

`listDestPaths` contains a list of all paths to `sFolderName` within `sStartPath` . `sDestPath` contains the first element of `listDestPaths` . `listDestFiles` contains a list of all files with name `sFileName` found within `listDestPaths` . `sDestFile` contains the first element of `listDestFiles` .

A semicolon separated list of file names in `sFileName` (like for `sFolderName`) is not supported.

Providing more than one folder together with a file name may cause overlapping results. But `listDestFiles` will not contain any redundant paths to files.

```
sStartPath  = r"D:\pathtest\ABC\DEF\GHI"
sFolderName = "ABC;DEF"
sFileName   = "FILE.txt"
sDestPath, listDestPaths, sDestFile, listDestFiles, sDestPathParent = ↩
    ↪ CString.DetectParentPath(sStartPath, sFolderName, sFileName)
```

Outcome

```
sDestPath       = D:/pathtest/ABC/DEF
listDestPaths   = ['D:/pathtest/ABC/DEF', 'D:/pathtest/ABC']
sDestFile       = D:/pathtest/ABC/DEF/GHI/FILE.txt
listDestFiles   = ['D:/pathtest/ABC/DEF/GHI/FILE.txt', 'D:/pathtest/ABC/FILE.txt']
sDestPathParent = D:/pathtest/ABC
```

In the last example we go one further level up (`pathtest`). Because of the file search is recursive, also files in parallel trees are found now.

```
sStartPath  = r"D:\pathtest\ABC\DEF\GHI"
sFolderName = "pathtest"
sFileName   = "FILE.txt"
sDestPath, listDestPaths, sDestFile, listDestFiles, sDestPathParent = ↩
    ↪ CString.DetectParentPath(sStartPath, sFolderName, sFileName)
```

Outcome

```
sDestPath       = D:/pathtest
listDestPaths   = ['D:/pathtest']
sDestFile       = D:/pathtest/ABC/DEF/GHI/FILE.txt
listDestFiles   = ['D:/pathtest/ABC/DEF/GHI/FILE.txt', 'D:/pathtest/ABC/FILE.txt', ↩
    ↪ 'D:/pathtest/RST/UVW/XYZ/FILE.txt']
sDestPathParent = D:/
```

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.

These filters can be e.g. used to select or exclude lines while reading from a text file. Or they can be used to select or exclude files or folders while walking through the file system. 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 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

All available filters are described in more detail in the interface description of `StringFilter` . Here we continue with some code examples.

Example 1

`sString` has to start with `sStartsWith` and has to contain `sContains` .

That's true. Therefore `StringFilter` 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)
```

Example 2

`sString` must not end with `sEndsNotWith` . But does. Therefore `StringFilter` returns `False` - even in case of other criterions are fulfilled.

```
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)
```

Example 3

`sString` must not contain `sContainsNot` . Because `bCaseSensitive` is `True` the spelling does not fit. Therefore `StringFilter` 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)
```

Example 4

`sString` must contain exactly two digits (postulated by regular expression based `sInclRegEx`). That's true. Therefore `StringFilter` 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)
```

Example 5

`sString` must contain exactly three digits (postulated by regular expression based `sInclRegEx`). That's not true. Therefore `StringFilter` returns `False` - even in case of other criterions are fulfilled.

```
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)
```

Example 6

Leading and trailing spaces are removed from the input string `sString` at the beginning. In this example the result is an empty input string. `bSkipBlankStrings` is set to `True` . In this case `StringFilter` immediately returns `False` and all other filters are ignored.

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

Example 7

The input string `sString` starts with a character that is defined to be a comment character (`sComment`). Therefore `StringFilter` immediately returns `False` and all other filters are ignored.

```
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)
```

Example 8

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

Example 9

In case of blanks around search strings have to be considered, a regular expression based filter has to be used (here `sInclRegEx`).

This is possible because the regular expression based filters `sInclRegEx` and `sExclRegEx` work **with the original value** of `sString` , and not with the shortened version with leading and trailing blanks removed! The shortened version is applied to standard filters only.

In this example `StringFilter` 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)
```

Example 10

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

Nevertheless `StringFilter` 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)
```

Example 11

In this example the semicolon is masked with `"\;"` and therefore part of the search string `sContains` - and not part of the syntax any more.

The meaning of `"beta\; not"` in this example is: The criterion is fulfilled in case of `"beta; not"` can be found. That's **not** `True` . Therefore `StringFilter` 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)
```

2.2.2 File access with CFile

The motivation for the `CFile` module contains two main topics:

1. Extended 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 independent 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.

Examples

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 return 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 - like an exception.
- `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 `Write()` after `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 only a single line from a file 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.

Internally `ReadLines()` uses the string filter method `StringFilter()` . All filter related input parameter of `ReadLines()` and `StringFilter()` are the same.

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 (the class object `oFile_1` still exists).

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 (therefore we also do not need the class object 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
```

2.2.3 Utilities

PrettyPrint

The idea behind the `PrettyPrint()` function is to resolve 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

The following Python code defines a composite data type and prints the content with `PrettyPrint()` :

```
dictTest = {'K1' : 'value',
            'K2' : ["A", 22, True, (33, 'XYZ')],
            'K3' : 10,
            'K4' : {'A' : 1,
                    'B' : 2}}

PrettyPrint(dictTest)
```

Result

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

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

For example the meaning of the fifth line of output

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

is:

- The type of input parameter `oData` is `dict`
- The dictionary contains 4 keys
- The current line gives information about the second key of the dictionary
- The name of the second key is `K2`
- The value of the second key is of type `list`
- The list contains 4 elements
- The current line gives information about the fourth element of the list
- The fourth element of the list is of type `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 `33`

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

Chapter 3

CComparison.py

3.1 Class: CComparison

Imported by:

```
from PythonExtensionsCollection.Comparison.CComparison import CComparison
```

The class `CComparison` contains mechanisms to compare two files either based on the original version of these files or based on an extract (made with regular expressions) to ensure that only relevant parts of the files are compared.

3.1.1 Method: Compare

Compares two files. While reading in all files empty lines are skipped.

Arguments:

- `sFile_1`
  / *Condition*: required / *Type*: str /
  First file used for comparison.
- `sFile_2`
  / *Condition*: required / *Type*: str /
  Second file used for comparison.
- `sPatternFile`
  / *Condition*: optional / *Type*: str / *Default*: None /
  Pattern file containing a set of regular expressions (line by line). The regular expressions are used to make an extract of both input files. In this case the extracts are compared (instead of the original file content).

Returns:

- `bIdentical`
  / *Type*: bool /
  Indicates if the two input files (or their extracts) have the same content or not.
- `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 4

CFile.py

4.1 Class: enFileStatiType

Imported by:

```
from PythonExtensionsCollection.File.CFile import enFileStatiType
```

The class `enFileStatiType` defines the sollowing file states:

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

4.2 Class: CFile

Imported by:

```
from PythonExtensionsCollection.File.CFile import 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 was successful or not.
- `sResult`
  / *Type*: str /
  The result of the computation of the method.

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

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

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

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.

Internally `ReadLines` uses the string filter method `StringFilter`. All filter related input parameter of `ReadLines` and `StringFilter` are the same.

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.

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 was successful or not.

- `sResult`

  / *Type*: str /

  The result of the computation of the method.

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

CFolder.py

5.1 Function: rm_dir_readonly

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

5.2 Class: CFolder

Imported by:

```
from PythonExtensionsCollection.Folder.CFolder import 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.

5.2.1 Method: Delete

Deletes the folder the current class instance contains.

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.

5.2.2 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.

5.2.3 Method: CopyTo

Copies the current folder to `sDestination`, that has to be a path to a folder **within** the source folder will be copied to (with it's original name),

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

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

Arguments:

- `sDestination`
  / *Condition*: required / *Type*: string /
  The path to destination folder
- `bOverwrite`
  / *Condition*: optional / *Type*: bool / *Default*: False /

  - In case of the destination folder already exists and `bOverwrite` is `True`, than the destination folder will be overwritten.
  - In case of the destination folder already exists and `bOverwrite` is `False` (default), than the destination folder 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.

Chapter 6

CString.py

6.1 Class: CString

Imported by:

```
from PythonExtensionsCollection.String.CString import CString
```

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

6.1.1 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`)

6.1.2 Method: DetectParentPath

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

6.1.3 Method: StringFilter

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.

These filters can be e.g. used to select or exclude lines while reading from a text file. Or they can be used to select or exclude files or folders while walking through the file system.

The following filters are available:

bSkipBlankStrings

- 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

Further details together with codde examples can be found within chapter **Description**, subsubsection **StringFilter**.

6.1.4 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 7

CUtils.py

7.1 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

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

7.2 Class: CTypePrint

Imported by:

```
from PythonExtensionsCollection.Utils.CUtils import CTypePrint
```

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.

7.2.1 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 8

Appendix

About this package:

Table 8.1: Package setup

Setup parameter	Value
Name	PythonExtensionsCollection
Version	0.11.5
Date	21.02.2023
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 9

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`	
0.8.0	28.06.2022
Module `CFolder`: Method: `CopyTo` added	
0.9.0	27.07.2022
History reworked (requires `GenPackageDoc` v. 0.26.0 at least)	
0.10.0	25.10.2022
Added `CComparison` module to compare two files (either based on original content or based on pattern)	
0.11.0	15.11.2022
Converted parts of the documentation from RST format to LaTeX format (to improve the layout).	

PythonExtensionsCollection.pdf

Created at 21.02.2023 - 14:58:41

by GenPackageDoc v. 0.39.0