Python Extensions Collection

Queckenstedt Holger (XC-CT/ECA3)

Feb 22, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

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

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

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

Path normalization

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

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

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

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

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

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

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

Module import

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

CFile

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

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

CString

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

CUtils

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

CHAPTER

TWO

EXAMPLES

2.1 File access with CFile

The motivation for the CFile module contains two main topics:

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

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

Define two variables with path and name of test files.

Under Windows:

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

Or under Linux:

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

The first class instance:

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

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

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

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

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

It's more save to implement in this way:

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

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

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

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

Write some content to file:

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

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

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

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

Some functions close the file handle (e.g. `ReadLines()`). Therefore sequences like `oFile_1.Write("...")`, `oFile_1.Readlines("...")`, `oFile_1.Write("...")` should be avoided - because the `Write()` after the `ReadLines()` starts the file from scratch and the file content written by the previous `Write()` calls is lost.

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

Append content to file:

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

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

Read from file:

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

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

Internally `ReadLines()` takes care about:

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

To avoid code like this

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

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

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

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

Let's assume the following:

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

This can be imlemented in the following way.

Read a subset of file content:

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

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

Close a file handle:

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

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

Additionally to that the file handle is closed implicitely:

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

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

```
del oFile_1
```

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

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

A file can be copied to another file.

Copy a file:

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

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

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

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

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

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

A file can be moved to another file.

Move a file:

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

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

A file can be deleted.

Delete a file:

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

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

1. *Explicitely do a deletion*

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

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

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

The optional parameter `bConfirmDelete` controls this behavior.

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

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

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

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

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

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

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

Avoid access violations

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

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

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

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

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

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

bSuccess, sResult = oFile_1.CopyTo(sFile_2)

```

(continues on next page)

(continued from previous page)

```
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
```

(*Last update: 26.01.2022*)

CHAPTER

THREE

CSTRING MODULE

class String.CString.**CString**

Bases: `object`

Contains some string computation methods like e.g. normalizing a path.

static DetectParentPath(*sStartPath=None*, *sFolderName=None*, *sFileName=None*)

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.

Args:

sStartPath (*string*)

The path in which to search for a parent folder

sFolderName (*string*)

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

sFileName (*string, optional*)

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

Returns:

sDestPath (*string*)

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 (*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 (*string*)

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 (*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 (*string*)

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

`static FormatResult`(*sMethod=''*, *bSuccess=True*, *sResult=''*)

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.

Returns the formatted result string.

`static NormalizePath`(*sPath=None*, *bWin=False*, *sReferencePathAbs=None*, *bConsiderBlanks=False*, *bExpandEnvVars=True*, *bMask=True*)

Method:

NormalizePath

Normalizes local paths, paths to local network resources and internet addresses

Args:

sPath (*string*)

The path to be normalized

bWin (*boolean; optional; default: False*)

If `True` then returned path contains masked backslashes as separator, otherwise slashes

sReferencePathAbs (*string, optional*)

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 (*boolean; optional; default: False*)

If `True` then the returned path is encapsulated in quotes - in case of the path contains blanks

bExpandEnvVars (*boolean; optional; default: True*)

If `True` then in the returned path environment variables are resolved, otherwise not.

bMask (*boolean; optional; 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 (*string*)

The normalized path (is `None` in case of `sPath` is `None`)

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

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 parameter:

sString

The input string that has to be investigated.

bCaseSensitive (*boolean, optional, default*: `True`)

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

bDebug (*boolean, optional, default*: `False`)

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

Examples:

Returns `True`:

```
bAck = CString.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)
```

Returns `False`:

```
bAck = CString.StringFilter(sString          = "Speed is 25 beats per minute",
                            bCaseSensitive   = True,
                            bSkipBlankStrings = True,
                            sComment         = None,
                            sStartsWith      = "Sp",
                            sEndsWith        = None,
```

(continues on next page)

(continued from previous page)

```
                                sStartsNotWith     = None,
                                sEndsNotWith       = "minute",
                                sContains          = "beats",
                                sContainsNot       = None,
                                sInclRegEx         = None,
                                sExclRegEx         = None)
```

Returns True:

```
bAck = CString.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)
```

Returns True:

```
bAck = CString.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)
```

Returns False:

```
bAck = CString.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)
```

Returns False:

```
bAck = CString.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}")
```

Returns False:

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

Returns False:

```
bAck = CString.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)
```

Returns False:

```
bAck = CString.StringFilter(sString          = "   Alpha is not beta; and beta␣
↪is not gamma   ",
                            bCaseSensitive    = True,
                            bSkipBlankStrings = True,
                            sComment          = None,
                            sStartsWith       = None,
                            sEndsWith         = None,
```

(continues on next page)

(continued from previous page)

```
                                  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.

This alternative solution returns `True`:

```
bAck = CString.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)
```

Returns `True`:

```
bAck = CString.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`:

```
bAck = CString.StringFilter(sString            = "Alpha is not beta; and beta is␣
↪not gamma",
```

(continues on next page)

(continued from previous page)

```
                            bCaseSensitive    = True,
                            bSkipBlankStrings = True,
                            sComment          = None,
                            sStartsWith       = None,
                            sEndsWith         = None,
                            sStartsNotWith    = None,
                            sEndsNotWith      = None,
                            sContains         = r"beta\; not",
                            sContainsNot      = None,
                            sInclRegEx        = None,
                            sExclRegEx        = None)
```

CHAPTER

FOUR

CUTILS MODULE

class `Utils.CUtils.`**`CTypePrint`**

Bases: `object`

`TypePrint`(*oData=None*, *bHexFormat=False*)

`Utils.CUtils.`**`PrettyPrint`**(*oData=None*, *hOutputFile=None*, *bToConsole=True*, *nIndent=0*, *sPrefix=None*, *bHexFormat=False*)

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.

Args:

oData (*any Python data type*)

A variable of any Python data type.

hOutputFile (*handle to a file opened for writing or appending; optional; default None*)

If handle is not `None` the content is written to this file, otherwise not.

bToConsole (*bool; optional; default: True*)

If `True` the content is written to console, otherwise not.

nIndent (*int; optional; default: 0*)

Sets the number of additional blanks at the beginning of every line of output (indentation).

sPrefix (*str; optional; default: None*)

Sets a prefix string that is added at the beginning of every line of output.

bHexFormat (*bool; optional; default: False*)

If `True` the output is printed in hexadecimal format (but strings only).

Returns:

listOutLines (*list*)

List of lines containing the prepared output

CHAPTER

FIVE

CFILE MODULE

class File.CFile.**CFile**(*sFile=None*)

Bases: `object`

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.

Append(*Content='', nVSpaceAfter=0, sPrefix=None, bToScreen=False*)

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

If `Content` is not a string, the `Write` method resolves the data structure (therefore `Content` can also be of type `list`, `tuple`, `set`, `dict`, `dotdict`).

Adds vertical space `nVSpaceAfter` (= number of blank lines) after `Content`.

Prints `Content` also to screen in case of `bToScreen` is `True` (default: `False`).

Returns `bSuccess` and `sResult` (feedback).

`Close()`

Closes the opened file.

Returns `bSuccess` and `sResult` (feedback).

`CopyTo(`*sDestination=None*, *bOverwrite=False*`)`

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`.

Returns `bSuccess` and `sResult` (feedback).

`Delete(`*bConfirmDelete=True*`)`

Deletes the current file.

bConfirmDelete

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` and `sResult` (feedback).

`GetFileInfo()`

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

Key **sFile**

Path and name of current file

Key **bFileIsExisting**

`True` if file is existing, otherwise not

Key **sFileName**

The name of the current file (incl. extension)

Key **sFileExtension**

The extension of the current file

Key **sFileNameOnly**

The pure name of the current file (without extension)

Key **sFilePath**

The the path to current file

Key **bFilePathIsExisting**

`True` if file path is existing, otherwise not

`MoveTo`(*sDestination=None*, *bOverwrite=False*)

Moves 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`.

Returns `bSuccess` and `sResult` (feedback).

`ReadLines`(*bCaseSensitive=True*, *bSkipBlankLines=False*, *sComment=None*, *sStartsWith=None*, *sEndsWith=None*, *sStartsNotWith=None*, *sEndsNotWith=None*, *sContains=None*, *sContainsNot=None*, *sInclRegEx=None*, *sExclRegEx=None*, *bLStrip=False*, *bRStrip=True*, *bToScreen=False*)

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.

T.B.C.

`Write`(*Content=''*, *nVSpaceAfter=0*, *sPrefix=None*, *bToScreen=False*)

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

If `Content` is not a string, the `Write` method resolves the data structure (therefore `Content` can also be of type `list`, `tuple`, `set`, `dict`, `dotdict`).

Adds vertical space `nVSpaceAfter` (= number of blank lines) after `Content`.

Prints `Content` also to screen in case of `bToScreen` is `True` (default: `False`).

Returns `bSuccess` and `sResult` (feedback).

class `File.CFile.`**`enFileStatiType`**

Bases: `object`

`closed = 'closed'`

`openedforappending = 'openedforappending'`

`openedforreading = 'openedforreading'`

`openedforwriting = 'openedforwriting'`

PYTHON MODULE INDEX