Python Module **pymanual**

Version pymanual 0.999

Dr. Harald Meyer auf'm Hofe
Kaiserslautern, Germany

August 2, 2006

Contents

1 Module pymanual

Module and program to extract documentation strings from modules and transform them into documents of various formats. These formats are currently:

1. HTB: A zipped directory of HTML files and files on content and index which are also understood by Microsofts help compiler. Refer to the wxWidgets project for information on the HTB-format and available viewers at *www.wxWidgets.org*.
2. PDF: Adobe's portable document format.
3. HTML: A directory of the same files that are used to compile an HTB document.
4. TEX: A LaTeX source file that may be used to compile DVI and PDF documents using the well known LaTeX word processor.

This document is a formatted version of the documentation strings in the **pymanual** module.

This file may be either used as a program or as an imported module.

When used as a program, the synopsis is

1. for producing a hyper text book (HTB, refer to the wxWidgets for viewers and class **FormatterHTB**):

   pymanual.py {**module_names**}
2. for producing HTML files (refer also to **FormatterHTB**):

   pymanual.py –html [-p LaTeX_preamble] [–preamble LaTeX_preamble] {**module_names**}
3. for producing PDF (refer also to **FormatterLaTeX**):

   pymanual.py –pdf [-p LaTeX_preamble] [–preamble LaTeX_preamble] {**module_names**}

   Please note that this requires the programs **pdflatex** and **makeindex**, common parts of most distributions of the LaTeX word processor, to be in the current paths where executables are searched. For details on the **LaTeX_preamble** refer to class **FormatterLaTeX**.
4. for producing LaTeX sources (refer also to **FormatterLaTeX**):

   pymanual.py –tex [-p LaTeX_preamble] [–preamble LaTeX_preamble] {**module_names**}

where

LaTeX preamble: This is an optional name of a file containing definitions to be pasted into the preamble of the produced LaTeX-file. Refer to class **FormatterLaTeX**.

module name: This is a sequence of module names specifying the modules to be documented by the resulting document.

In case of producing a documentation of a single module, the output will be written into a file or directory whose base name will be the name of the module completed by a suffix describing the type of the document. As example consider the production of a documentation for this module:

- **pymanual.py pymanual** produces a file **pymanual.htb**.
- **pymanual.py –html pymanual** produces a directory **pymanual** containing HTML files (among others).
- **pamanual.py –pdf pymanual** produces a file **pymanual.pdf**. This required **pdflatex** to be in the execution path.

- **pamanual.py –tex pymanual** produces a file **pymanual.tex**.

In case of collecting the documentation of more than one module into one document, the base name of the resulting document is **refman** completed by an appropriate suffix.

For usage as an imported module refer to the functions **pymanual_latex** and **pymanual_htb**.

1.1 State Of Implementetion

As indicated by the version identifier, the current version of **pymanual.py** can be considered as a fairy complete but weakly tested implementation. The major test case has been the documentation of the program itself.

1.2 Motivation

Professionally, I use the programming languages C/C++, C# and JAVA. On these programming languages, the tool **doxygen** is available for in-source-documentation. Being able to produce comprehensible documentation of classes and functions directly within the source code proved to be extremely valuable.

- In the specification phase, motivation and specification of a design can be attached to declarations of classes and functions before implementation. However, this can also be achieved by producing extra documents for specification. Nevertheless, in-source-documentation eases maintenance of this information in order to keep it useful.
- In-source-documentation proved to ease maintenance and extension of implementations:
  - A maintained specification of classes and functions adds the necessary redundancy to decide which implementation caused an erroneous behaviour of the implemented system: Is there an implementation fault in a particular method of a function or has it been used mistakenly?
  - A comprehensible specification also serves as a manual and eases code reusing.

Python attaches documentation strings to each class, function, and method to serve mainly as a manual for using it. Documentation strings can be used quite like man-pages on UNIX systems and program **pydoc** produces manuals in HTML from these strings. Unfortunately, even manuals produced by **pydoc** are simple presentations of plain text lacking even simple text structures like sections and item lists. Python code can hardly be separated from free text. Following references to related implementations (other classes or functions) is not supported by hyperlinks.

pymanual.py is yet another approach to extend the usability of documentation strings. The program recognizes each occurance of a function name or a class name and produces hyperlinks to the corresponding documentation. **pymanual.py** allows the specification of section, item lists, enumerations, and description lists. Inlined code fragments may be highlighted. The program also recognizes parts of the documentation string that obviously have been added for use by module **doctest**.

pymanual does not produce any diagrams but

- documentations of classes contain a synopsis section providing hyperlinks to inherited and inheriting classes, and
- documentations of class methods contain a synopsis section on signature and occurances of methods of similar names in other documented classes. The latter feature has been added to reflect the fact that **python** does not use strong types. From a software refactorers view, each implemented method is similar to an interface in JAVA or C#. An object's ability to serve as an argument of a method or function depends exclusively on the availablity of the required methods and member variables (in contrast to type compatibility in strongly typed languages). So, hyperlinks from method *A* to methods of equal name provide a strong hint to recognize classes of objects that may be used in the same role as implementors of *A*.

Modules, classes, and funtions providing **pymanual** documentation may also specifiy some limited degree of meta information on authors, version, and copyright that will be displayed in the documentaton 1.4 (page 7) However, like **pydoc** this program requires the imported modules to be compilable (since modules to be documented will be imported before producing any ouput).

1.3 Formatting Text

Text formatting with **pymanual.py** aims mainly at achieving two goals:

1. Availability of the same opportunities for structuring texts as they are available in books plus the option to inline Python source code plus the option to present lines of code, either to be processed by module **doctest** or not.
2. Documentation shall also be comprehensible from the plain text source. So, the task of documenting items by use of **pymanual.py** results in at least two documents:
   (a) One document in PDF or HTB providing a rich set of opportunities for structuring and highlighting text.
   (b) A plain text document to be processed by **pydoc** or the internal **help** command using only the original line breaks as text layout.

For achieving both goals, **pymanual.py** rather uses *wikipedia* like short formatting characters to describe the text format of a text item than using markup tags like in HTML etc. To restrict the effect of missing closings of formatting tags (and for the sake of easier implementation), tags are only valid within the same line of the documentation source (the __**doc**__ string).

1.3.1 Structering Texts

Linebreaks in the original plain text will not break lines in the richly formatted documentation. As an exception of this general rule, empty lines introduce new text paragraphs in the richly formatted documentation.

Use the lower than (<) and the larger than characters to embrace a single word that shall be highlighted as a code fragment (a file, class or function name). Use two lower than and greater than characters to embrace code fragments containing more than one word within the same line of the plain text source of the documentation. Note, that names of also documented modules, classes, and functions will recognized by the system. So, embracing such words is optional but recommended, to enhance comprehensibleness of the plain text. Inlined code fragments are usually presented in bold letters. Inlined code fragments are often printed in true type letters. However, using bold letters instead results into a much more harmonic text layout.

Analogously, "<!" and "!>" might be used to embrace emphasized text passages. These passages are usually printed in italics.

pymanual supports item lists, enumerations, and description lists. Use the star (*) to introduce an item of an item list as the first character of a new line in the original plain text. The double cross (#) specifies a new item of an enumeration. To specify a new item in a description list, type the name of the description (a single word) followed by a colon (:). If the name of the description consists of more than one word, use the underscore (_) as separator. The rest of the line in the plain text is considered to hold text describing the item. If the next lines start with a colon, they will also be added to the description of the item.

Each item may contain additional item lists, enumerations, or description lists. To start an encapsulated item list, start the line in the plain text source with a colon directly followed by s start (:*). Use ":#" to start a new enumeration or "item_name:" to start a description list start with a description on "item name".

Example:

- an **item covering two lines** in the original plain text documentation.

  1. encapsulated enumeration item 1.
  2. encapsulated enumeration item 2.

     description item1: This is a part of the enumeration.

  description item2: This is a part of the item list.

description item3: This is indention level 0.

description item4: This also.

Plain text source of the example:

```
<!Example:!>
* an <<item>>
: <<covering two lines>> in the original plain text documentation.
:# encapsulated enumeration item 1.
:# encapsulated enumeration item 2.
::description_item1: This is a part of the enumeration.
:description_item2: This is a part of the item list.
description_item3: This is indention level 0.
description_item4: This also.
```

1.3.2 Long Code Fragments And Preformatted Text

In addition to inlined code fragments that typically consist of single program symbols or short expressions, program documentations usually contain longer code fragments to present source code examples or preformatted text to show for instance text examples for configuration files etc.

Preformatted Text :

```
 This is some preformatted text.
      The content is presented in true type letters.
    Preceed each line containing preformatted text with a bar '|'.
```

Original plain text of the example above:

```
| This is some preformatted text.
|      The content is presented in true type letters.
|    Preceed each line containing preformatted text with a bar '|'.
```

Python Source Code : Source code occurs in Python documentations for two purposes:

1. Present source code examples, and
2. input for the **doctest** module.

pymanual.py supports both by a special mode you can turn on typing a Python prompt as the first non-blank characters of a line in the plain documentation. This mode is the only one that spans over more than one line and has to be left by an empty line. Example:

```
>>> 2+2   # a Python expression
4
>>>   # in this mode you have to provide the result of an expression
>>>   # since module doctest looks for prompts in the documentation
>>>   # an will report errors if results are missing.
>>> context=FormattingContext(pymanual)
>>>   # names of modules, classes, functions, and methods will also be
```



```
>>>   # recognized in this mode.
>>>
>>>   # leave this mode by an empty line in the plain text documentation.
```

In plain text, this example looks as follows:

```
>>> 2+2 # a Python expression
4
>>> # in this mode you have to provide the result of an expression
>>> # since module doctest looks for prompts in the documentation
>>> # and will report errors if results are missing.
>>> context=FormattingContext(pymanual)
>>> # names of modules, classes, functions, and methods will also be
>>> # recognized in this mode.
>>>
>>> # leave this mode by an empty line in the plain text documentation.
```

If you intend to use **doctest**, you may want to present code snippets that you do not want to be evaluated by **doctest**. In such cases you should use "|>>" instead of the standard Python prompt ">>>". This keyword will result into exactly the same formatted documentation but will be ignored by **doctest**.

```
|>> 2+2   # a simple Python expression
5
|>>   # the example above presents an obvously wrong result. however,
|>>   # it will not be tested anyway.
|>> pymanual_htb(['pymanual'], '.','HTB', FormatterHTB.m_englishPattern, FormatterHTB.m_englishTexts,
FormatterHTB.m_englishTitle [])
|>>   # modules, functions, and classes will also be recognized in this mode.
```

1.3.3 Sectioning, Referencing, Indexing

Longer texts (like this one) should be organized in sections. **pymanual.py** supports three levels of sections. Embrace the headline of a section in the plain text documentation with "==", "===", or "====" to start a new section. For instance, the headline of this section has been produced by

```
=== Sectioning, Referencing, Indexing ===
```

Please note, that this three levels have to be mapped to sectioning levels provided by the used rich format. For instance, in LaTeX documents level, "===" on modules is a **subsubsection** and on methods it is a **paragraph**.

A documentation may contain cross references to sections within the same documentation string. Therefore, a label has to be attached to a section. Labels are strings composed of letters and numbers serving as an identifier for a section. For instance, this section has the label **refs**. This label has been defined in the headline:

```
refs:=== Sectioning, Referencing, Indexing ===
```

Now, each text within this documentation string may refer to this section 1.3.3 (page 6) using the markup tag **:ref[refs]**. Please refer to 1.5 (page 7) for current limitations of tags in **pymanual.py** in general and limitations of the **ref** markup tag in particular.

For the sake of comprehensibleness of the plain text documentation, **pymanual.py** provides only few tags, in fact only two. The other one is for producing a hierarchical index. **pymanual.py** produces an hierarchical index on modules, classes, functions, and methods automatically. However, documentation on source code may introduce additional notation for instance on domain concepts that should be listed in the index. This can be achieved by use of tag **index**. This paragraph contains the tag **:index[index|example]**. As a consequence, the hierarchical index of this document comprises the key word "index" with aspect "example" associated with a link to this paragraph.

1.4 Meta information

pymanual.py analyses three sources of information. Two of these sources have been the issue of the previous sections.

1. information on classes, methods, and functions as provided by the reflective libraries of Python.
2. documentation strings as attached to modules, classes, methods, and functions.

Additionally, **pymanual.py** searches the main dictionary of modules, classes, methods, and functions for the attributes __**author**__, __**copyright**__, and __**version**__. The data associated with these attributes will be presented in the documentation depending on the produced format of the richly formatted document.

Information on author, copyright, and version is inherited as usual in Python. Usually, information on these topics will only be presented if it changes. For instance in this file, only the **pymanual** module provides information on author, copyright, and version. So, this information will not be listed for functions and classes.

- __**author**__ may be associated with a string describing the author of the documented entity. For instance, module **pymanual** defines the author "Dr. Harald Meyer auf'm Hofe, Kaiserslautern, Germany".
- __**copyright**__ may be associated with a string describing information on the copyright. For instance, module **pymanual** defines the copyright **"(c**) Dr. Harald Meyer auf'm Hofe 2005-2006".
- __**version**__ may be associated with arbitrary data whith <**str(**)> producing an informative representation. Module **pymanual** defines its version by float **0.999**.

1.5 Known Bugs

1. Problems with tags **index** and **ref**: do not miss introducing blank.
2. Problems with tag **ref**: Works in HTB mode currently only within the same context. Probably this problem will be fixed somewhere in the future introducing a **longref** tag that also receives the target context as an argument. The system does not test whether the target of a reference actually exists. So, it produces blind references without warning. This is due to the fact that the design of this program strictly follows a single pass approach. It does not produce any tables for references and labels to ensure consistency.
3. Insertion in style codeline (mode for doctest): Inserting blanks will be inserted before the **...** prompt.

1.6 History

- **0.999** Mai 2006: Fixes. Access to member attributes by **getattr()**. Better support for directly imported classes.
- **0.99** Feb. 2006: Version with HTB and LaTeX output. Some error fixes on argspec interpretation etc.
- **0.9** Dec. 2005: Intial version with PDF and LaTeX output

1.7 License

EMail: harald.mah@gmx.de

2 Class Formatter

Extending classes: FormatterHTB, FormatterLaTeX

Base class of formatters. Specializations of this class act like an output file that writes standard text as well as some additional formatting directives. The different formatters differ in the way they quote special characters in standard text passages and in the way they use and **write** formatting directives.

This standard formatter currently does not provide any output. This implementation might be extended in the future to implement a **man** function similar to the **help** function but with evaluation of formatting directives.

2.1 Function close

Synopsis: **close(***self***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Write closing statements and **close** file. This class provides a default implementation that simply closes **m_dest**.

2.2 Function closeContext

Synopsis: **closeContext(***self, context***)**

Related implementations in classes: FormatterHTB

This is called by **pymanual** if all documentation to the object denoted by **context** has been written.

2.3 Function headline1

Synopsis: **headline1(***self, context, title***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This creates a new section header level 1 within the documentation of the object as described by the context, an instance of **FormattingContext**.

2.4 Function headline2

Synopsis: **headline2(***self, context, title***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This creates a new section header level 2 within the documentation of the object as described by the context, an instance of **FormattingContext**.

2.5 Function headline3

Synopsis: **headline3(***self, context, title***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This creates a new section header level 3 within the documentation of the object as described by the context, an instance of **FormattingContext**.

2.6 Function write

Synopsis: **write(***self, context, standard_text, underscoreIsBlank=False, quotState='left'***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Write string with that characters quoted that have a special meaning in LaTeX.

- If **underscoreIsBlank** is **True**, then the underscore is replaced by a blank character. Otherwise, the underscore will be quoted.
- **quotState** should be either "left" or "right" (where every string differing from "left" is consodered to mean "right"). This argument defines, whether the next quotation mark should open ("left") or **close** ("right") a quotation.
- Parameter **context** is an instance of class **FormattingContext** describing the current context of the documentation.

Result: The quotState after writing **standard_text**.

2.7 Function write_class_header

Synopsis: **write_class_header(***self, modname, classname, nom, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Introduces a new section that describes the class of name classname within module named modname. Argument nom provides the number of modules to be documented. The context consists simply of modname and classname.

metainfo is an instance of class **MetaInfo** holding meta info on the class.

2.8 Function write_class_synopsis

Synopsis: **write_class_synopsis(***self, context, base_classes, extending_classes, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Write the synopsis (base classes and specializations) for the class as described by context (which is supposed to be a **FormattingContext**). base_classes and extending_classes are lists of **FormattingContext**. metainfo is an instance of **MetaInfo**.

2.9 Function write_codelinebreak

Synopsis: **write_codelinebreak(***self, context***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Same purpose as **write_linebreak()** but assume that this is within a codeline environment.

2.10 Function write_codelineinsertion

Synopsis: **write_codelineinsertion(***self, context***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This indicates an insertion of a codeline, an introducing blank of a codeline.

2.11 Function write_fn_synopsis

Synopsis: **write_fn_synopsis(***self, context, argspec, base_implementations, related_implementations, unrelated_implementations, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Print the synopsis of a function element.

context: This is an instance of **FormattingContext** describing the object that is currently documented. **context.memberName()** is the name of the function.

argspec: The description of formal parameters in a format as returned by **inspect.getargspec()**.

base implementations: A list of context descriptions (class **FormattingContext**) representing base classes implementing the same method name.

related implementations: A list of context descriptions (class **FormattingContext**) representing classes connected to the class in **context** by inheritance and implementing the same method name.

unrelated implementations: A list of context descriptions (class **FormattingContext**) representing classes connected without an inheritance link to the class in **context** but implementing the same method name.

metainfo: An instance of **MetaInfo**.

2.12 Function write_function_header

Synopsis: **write_function_header(***self, context, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Generate a new section for describing the method funname of class classname in module modname. Parameter classname may be **None** to indicate a global function. Argument nom provides the number of modules to be documented.

The context is defined by modname, classname and funname.

metainfo is an instance of class **MetaInfo** holding meta info on the class.

2.13 Function write_header_fn_section

Synopsis: **write_header_fn_section(***self, modname***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Global module functions are usually written into a separate section. This function is called before documenting module functions to generate this section.

2.14 Function write_index

Synopsis: **write_index(***self, context, indexentries, mainIndex***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

An hierarchical index is expected to point at the position in the text where this is to be printed. Argument indexentries is a list of index entries, where the final entry is the most specific term.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

mainIndex is **True** iff this index shall be listed as the main index to this topic.

2.15 Function write_item

Synopsis: **write_item(***self, context, modeidentifier, itemname=None***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This is to **write** items in the modes “itemize”, “enumerate”, or “description” (as stated by argument modeidentifier). Argument itemname is a string used to provide a name for this item (cuurently only relevant for “description”). Note, that itemname contains underscores “_” instead of white spaces.

Refer also to **write_mode()**.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

2.16 Function write_label

Synopsis: **write_label(***self, context, labelname***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Create a label or an anchor, a possible target for **write_ref()**. **labelname** should be qualified with **FormattingContext.get_label()** in the same way as in **write_ref()**.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

2.17 Function write_linebreak

Synopsis: **write_linebreak(***self, context***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

The text after this statement is supposed to start in a new line.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

2.18 Function write_mode

Synopsis: **write_mode(***self, context, mode, close=False***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This is the method that prints the necessary output to open or **close** resp. one of the modes “verbatim”, “itemize”, “enumerate”, “description”, “code”, “codeline”, or “codelinerem”.

Refer also to **write_item()**.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

2.19 Function write_multi_module_header

Synopsis: **write_multi_module_header(***self, modname, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This creates a module section concerning module named modname in a document describing several modules. The context consists simply of the module.

2.20 Function write_ref

Synopsis: **write_ref(***self, context, labelname, targetContext=None***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Create a reference to label labelname in the module as given by targetContext. labelname can be expected to be unique within the same context. The targetContext is planned to be used in a (not yet implemented) **longref** tag for references to labels in other contexts. If this is **None** than use context as qualificator.

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

If modname is not **None**, this modname is used as qualificator for the referred label. Otherwise, **FormattingContext.moduleName()** is used.

2.21 Function write_refToContext

Synopsis: **write_refToContext(***self, context, refTo, refToIsMembervar=False***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

context: The context where the documentation has to be written to (instance ofFormattingContext).

refTo: Also an instance of **FormattingContext** that represents the object whose name has to be written here occasionally with a reference or hyperlink to its documentation.

refToIsMembervar: This is true iff argument refTo refers to a member variable (a **FormattingContext.className()**) is given) or a globally defined variable (the class name is not given).

2.22 Function write_single_module_header

Synopsis: **write_single_module_header(***self, modname, metainfo***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

This creates a module section for a document describing a single module of name modname. The context consists simply of the module.

metainfo is an instance of class **MetaInfo** holding meta info on the class.

2.23 Function write_undocumented

Synopsis: **write_undocumented(***self, context***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Argument context describes an undocumented element.

2.24 Function write_vars

Synopsis: **write_vars(***self, context, vars, nom***)**

Related implementations in classes: FormatterHTB, FormatterLaTeX

Generate a new section for describing variables defined in a module or a class. Argument **vars** is a dictionary that maps a value to the name of a member variable in class classname. This is only called if argument **vars** is not empty. Globals are variables of a module. Member variables are variables of a class.

2.25 Member Formatter

m_dest = None

3 Class FormatterHTB

Base classes: Formatter

This class is the currently available output to HTML and hyper text books (HTB). Text layout is based on page layout pattern **m_pattern** (set by __**init**__()) where certain tags are replaced by generated text. All these tags are introduced by a dollar sign (which is here in the documentation followed by a blank to prevent tag expansion).

$ NEXTLINK: Gets replaced by the URL to the next HTML file in the document (not available in the title page).

$ NEXTTITLE: Gets replaced by the title of the next HTML file in the document (not available in the title page).

$ PREVLINK: Gets replaced by the URL to the previous HTML file in the document (not available in the title page).

$ PREVTITLE: Gets replaced by the title of the previous HTML file in the document (not available in the title page).

$ UPLINK: Gets replaced by the URL to the next higher level HTML file (not available in the title page).

$ UPTITLE: Gets replaced by the title of the next higher level HTML file (not available in the title page).

$ TITLE: The title of the current element (not available in the title page).

$ MODNAMES: A comma separated list of the names of the modules to be documented. Without links.

$ MODLIST: An item list or table of the modules to be documented includeing hyperlinks.

$ PARTLINKS: A list of hyperlinks to parts of the current context. The parts of the empty context are the modules to be documented. The parts of a class are the methods. A method or a function has no parts, so this will be replaced by the empty string.

$ TODAY: Date in English format.

$ TOC: Table of contents: An enumeration of all pages.

$ HEUTE: Date in German format.

$ DOCUMENTATION: The documentation of the current element in HTML format (not available in the title page).

$ AUTHORS: The author of the element. This may be an empty string.

$ COPYRIGHT: A string describing copyright info. This may be an empty string.

$ VERSION: A string describing the version. This string may be empty.

Users may add general remarks to an overall project (including several modules) providing a particular title page by **__init__()** and optionally additional HTML pages as additional files. Note, that functions, classes, and methods are put in files named according to **FormattingContext.get_label()+'.html'**.

Some of the members of this class represent actual state. Others provide default parameters. Members contributing to the state:

m_outputType: Defines the type of desired output. Currently allowed are the strings "HTB" and HTML.

m_outputDir: Defines the directory where to save the desired output.

m_listofmodules: The list of names of the modules to be documented.

m_tempDir: Holds the pathname to a temporary directory if required.

m_filename: Holds the filename of the desired output file without suffix.

m_index: If this is not **None** then this is a list of tuples (indices, label) for the index file (suffix **.hhk** in input for Microsoft help compiler or HTB archives). This map is filled on generating documentations.

m_indexNo: A counter for the targets of indices from **m_index**. This counter is used to produce unique labels for the indices to point at.

m_openDocuments: Is a mapping of context (instance of **FormattingContext**) to strings representing the content of the **$DOCUMENTATION** tag for the object represented by the key.

m_itemsToClose: Unfortunately, the interface from code analysis does not provide any menas to indicate the end of an item element. However, items can be expected to end with either the closing of a list element or the start of a new item. So, this member provides the necessary state to **close** any open list element providing a stack on list elements to be closed in appropriate situations. This stack holds the tag for closing the item. This is done on method **closeItem()**.

m_pattern: The general pattern for HTML pages.

m_texts: Is a list of predefined texts.

m_titlePage: The pattern of the title page. This page is only generated on documenting more than one module.

m_additionalFiles: A list of filenames that should be integrated into the resulting documentation because they serve as source of multimedia data or as destination of hyper links. Note, that these files will be added to the resulting document with their base names. So, hyper links should refer to the base names of these files and base names shall not be shared among additional files.

The following members provide hints for calling the class and can be requested statically.

m_englishPattern: A simple **m_pattern** for English documentation.

m_germanPattern: A simple **m_pattern** for German documentation.

m_englishTexts: An English version of **m_texts**.

m_germanTexts: A German version of **m_texts**.

m_englishTitle: An English pattern for the title page.

m_germanTitle: A German pattern for the title page.

3.1 Function __del__

Synopsis: **__del__(***self***)**

Call **close()** this.

3.2 Function __init__

Synopsis: **__init__(***self, listofmodules, output_dir='.', output_type='HTB', pattern=None, texts=None, titlePage=None, additionalFiles=[]***)**

Initialization of the formatter. This initializes **m_pattern** with **pattern**, **m_texts** according to **texts**, and **m_titlePage** with titlePage. If these arguments are **None**, then this method assigns the defaults which are **m_englishPattern**, **m_englishTexts**, and **m_englishTitle**. Argument **additionalFiles** is a mostly empty list of additional files that belong to the documentation (for instance images for buttons that are used in **pattern** or logos etc). These files are copied into the HTML directory or HTB file resp. Please note, that these files shall have different base names because they get copied into the same directory.

The overall procedure is as follows:

1. set **m_outputType**, **m_outputDir**, m_listofmodules, and **m_filename**. **m_filename** is "refman" if **listofmodules** contains more than one name. Otherwise, filename is equal to the name of the module to be documented.
2. generate a new directory named **m_filename** in **output_dir** (HTML) or a temporary directory (HTB).
3. store name of the eventually generated temporary directory in **m_tempDir**.
4. generate the title file **index.html**.

3.3 Function addIndex

Synopsis: **addIndex(***self, names, label***)**

Add a new entry to the **m_index**.

mode: "Local" or "See Also"

names: A list of strings. This is the entry in the index.

label: This is the hyperlink where to point to.

3.4 Function close

Synopsis: **close(***self***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

Actually generates the HTML pages of the documentation taking for each entry in **m_openDocuments** a pattern from **m_pattern** and replacing all pattern tags like **$DOCUMENTATION**. This method also generates **"index.hhp"**. If **m_outputType** is "HTB", then all HTML pages are collected in **m_tempDir**, and this method produces a zip archive of all generated files in **m_outputDir** of name **m_filename** +**.htb**.

3.5 Function closeContext

Synopsis: **closeContext(***self, context***)**

Base implementations in classes: Formatter

This is called by **pymanual** if all documentation to the object denoted by **context** has been written.

3.6 Function closeItem

Synopsis: **closeItem(***self, context***)**

This is called on starting a new item or on leaving a list element. The method looks at **m_itemsToClose** for an item element to be closed and closes it reducing stack **m_itemsToClose**.

3.7 Function getContextTitle

Synopsis: **getContextTitle(***self, context, argContext***)**

This returns a title string for the instance **argContext** of **FormattingContext**. This is basically **FormattingContext.get_name(context**) but also classifying **argContext** as a representation of a module, a class or a method.

3.8 Function headline1

Synopsis: **headline1(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.9 Function headline2

Synopsis: **headline2(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.10 Function headline3

Synopsis: **headline3(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.11 Function initContext

Synopsis: **initContext(***self, context***)**

Make sure that **m_openDocuments** has a document for the provided context.

3.12 Function replacePatternTags

Synopsis: **replacePatternTags(***self, pattern, context=None***)**

This method returns a string where all the pattern tags in string **pattern** are replaced by the available information. This information is partially derived from the entry in **m_openDocuments** concerning **context**, a **FormattingContext**. If **context** is **None**, only the tags available to the title page will be replaced. Prerequisite: **m_openDocuments** has been filled.

This method is used in **close()** (with context argument) and in **__init__()** (without context argument).

3.13 Function write

Synopsis: **write(***self, context, standard_text, underscoreIsBlank=False, quotState='left'***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

This currently provides only transformations for German “Umlaute”, ligatures, and the special characters <, >, &, and “. The quotstate is irrelevant to this output format. Also implemented: Special treatment of backslash for quoting “, underscore, and itself with reference to argument underscoreIsBlank.

3.14 Function write_class_header

Synopsis: **write_class_header(***self, modname, classname, nom, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.15 Function write_class_synopsis

Synopsis: **write_class_synopsis(***self, context, listofmodules, base_classes, extending_classes, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.16 Function write_codelinebreak

Synopsis: **write_codelinebreak(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.17 Function write_codelineinsertion

Synopsis: **write_codelineinsertion(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.18 Function write_fn_synopsis

Synopsis: **write_fn_synopsis(***self, context, argspec, base_implementations, related_implementations, unrelated_implementations, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.19 Function write_function_header

Synopsis: **write_function_header(***self, context, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.20 Function write_header_fn_section

Synopsis: **write_header_fn_section(***self, modname***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.21 Function write_index

Synopsis: **write_index(***self, context, indexentries, mainIndex=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

Generates an index entry in **m_index**.

3.22 Function write_item

Synopsis: **write_item(***self, context, modeidentifier, itemname=None***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

This method also adds to **m_itemsToClose**.

3.23 Function write_label

Synopsis: **write_label(***self, context, labelname***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.24 Function write_linebreak

Synopsis: **write_linebreak(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.25 Function write_mode

Synopsis: **write_mode(***self, context, mode, close=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.26 Function write_multi_module_header

Synopsis: **write_multi_module_header(***self, modname, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.27 Function write_ref

Synopsis: **write_ref(***self, context, labelname, targetContext=None***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

Produce a link to generate a hyper link to label **labelname** in **targetContext**. If target is **None**, than assume **labelname** to be a label in the current context **context**.

FixMe: Currently, argument **targetContext** is not used in the program. So, you can only produce links within the same context.

3.28 Function write_refToContext

Synopsis: **write_refToContext(***self, context, refTo, refToIsMembervar=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.29 Function write_single_module_header

Synopsis: **write_single_module_header(***self, modname, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.30 Function write_undocumented

Synopsis: **write_undocumented(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.31 Function write_vars

Synopsis: **write_vars(***self, context, vars, nom***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterLaTeX

3.32 Member FormatterHTB

m_additionalFiles = []

m_englishTexts =

```
[ 'Base classes:',
  'Extending classes:',
  '<sup><i>(cf.)</i></sup>',
  '<i>Undocumented.</i>',
  'Module',
  'Class',
  'Method',
  'Member of class',
  'Globals of module',
  'Function',
  'Functions',
  'Synopsis:',
  'Base implementations in classes:',
  'Related implementations in classes:',
  'Unrelated implementations in classes:',
  'Documentation to Python module',
  'Documentation to Python modules',
  'Contents',
  'none']
```

m_pattern = ”

m_englishTitle =

```
'''

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
<title>$MODNAMES</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
</head>
<body bgcolor=white link=#D812D8 vlink=#D812D8>
<center>
<font color="#D812D8">Contents</font>
</center>
<hr color=#D812D8>
<h1>Python modules: $MODNAMES</h1>

<a name="toc"><h2>List of documented elements:</h2></a>
$TOC

<br>
<hr color=#D812D8>
<font size=-2 color="#D812D8"><i>
<div align="left">$COPYRIGHT </div> <div align="right">$VERSION</div><br>
Generated by <b>pymanual</b> (c) 2005-2006 on $TODAY.
</i></font>
</body>
</html>

'''
```

m_germanTitle =

```
'''

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
<title>$MODNAMES</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
</head>
<body bgcolor=white link=#D812D8 vlink=#D812D8>
<center>
<font color="#D812D8">Inhalt</font>
</center>
<hr color=#D812D8>
<h1>Python-Module: $MODNAMES</h1>

<a name="toc"><h2>Verzeichnis der dokumentierten Elemente:</h2></a>
$TOC

<br>
<hr color=#D812D8>
<font size=-2 color="#D812D8"><i>
<div align="left">$COPYRIGHT </div> <div align="right">$VERSION</div><br>
Erzeugt durch <b>pymanual</b> (c) 2005-2006 am $HEUTE.
</i></font>
</body>
</html>

'''
```

m_indexNo = 0

m_germanTexts =

```
[ 'Oberklassen:',
  'Unterklassen:',
  '<sup><i>(vgl.)</i></sup>',
  '<i>Keine Dokumentation vorhanden</i>',
  'Modul',
  'Klasse',
  'Methode',
  'Merkmale der Klasse',
  'Globale Variablen des Moduls',
```

```
        'Funktion',
        'Funktionen',
        'Synopsis:&Uuml;berl&auml;dt Implementationen in den folgenden Klassen:',
        'Gleichnamige Methoden innerhalb der Vererbungshierarchie in den Klassen:',
        'Gleichnamige Methoden au&szlig;erhalb der Vererbungshierarchie in den Klassen:',
        'Dokumentation des Python-Moduls',
        'Dokumentation der Python-Module',
        'Inhaltsverzeichnis',
        'keine']
```

m_outputDir = '.'

m_itemsToClose = []

m_outputType = 'HTB'

m_texts = []

m_filename = None

m_englishPattern =

```
    '''

    <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
    <html>
    <head>
    <title>Doc.: $TITLE</title>
    <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
    </head>
    <body bgcolor=white link=#D812D8 vlink=#D812D8>
    <center>
    <a href="index.html">Contents</a>   $UPTITLE <br>
    &lt;&lt;$PREVTITLE&lt;&lt;   &gt;&gt;$NEXTTITLE&gt;&gt;
    </center>
    <hr color=#D812D8>

    $DOCUMENTATION

    <br>
    <hr color=#D812D8>
    <a name="parts"><h2>Elements:</h2></a>
    $PARTLINKS

    <br>
    <hr color=#D812D8>

    <font size=-2 color="#D812D8"><i>
    <div align="left">$COPYRIGHT </div> <div align="right">$VERSION</div><br>
    Generated by <b>pymanual</b> (c) 2005-2006 on $TODAY.
    </i></font>
    </body>
    </html>

    '''
```

m_germanPattern =

```
    '''

    <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
    <html>
    <head>
    <title>Dokumentation: $TITLE</title>
    <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
    </head>
    <body bgcolor=white link=#D812D8 vlink=#D812D8>
    <center>
    <a href="index.html">Inhalt</a>   $UPTITLE  
    <br> &lt;&lt;$PREVTITLE&lt;&lt;   &gt;&gt;$NEXTTITLE&gt;&gt;
    </center>
    <hr color=#D812D8>

    $DOCUMENTATION

    <br>
```

```
<hr color=#D812D8>
<a name="parts"><h2>Elemente:</h2></a>
$PARTLINKS

<br>
<hr color=#D812D8>

<font size=-2 color="#D812D8"><i>
<div align="left">$COPYRIGHT </div> <div align="right">$VERSION</div><br>
Erzeugt durch <b>pymanual</b> (c) 2005-2006 am $HEUTE.
</i></font>
</body>
</html>

'''
```

m_index = []

m_titlePage = ”

m_tempDir = None

m_openDocuments = {}

4 Class FormatterLaTeX

Base classes: Formatter

This class class is the formatter producing text documents.

Members for configuration:

- **m_german_preamble** holds a preamble with translations of the internally used strings into German.
- **print_verbatim** is true iff all output is to be put directly to the destination without care on quoting TeX special characters.
- **m_filename** is the name of the destination file (probably in a temporary directory) without path name.
- **m_tempdir** is the name of the used temporary directory.

Users may use particular preamble files to configure the output. For instance, redefine LaTeX command **generalremarks** to put some additional remarks before the first section on modules.

4.1 Function __del__

Synopsis: **__del__(***self***)**

Call **close()** this.

4.2 Function __init__

Synopsis: **__init__(***self, listofmodules, output_dir='.', output_type='TEX', latex_preamble=''***)**

Initialize m_dest with an output file object to be used as destination for all output. Argument latex_preamble can be used to provide some additional LaTeX statements for the preamble of the resulting document. Argument listofmodules is the list comprising the names of the modules to be documented. This is for instance used to produce an initial implementation of the macro **printtitle** for the titlepage and for page headers and footers. **listofmodules** is also used to generate a filename:

- if only one module is documented, the name of this module is also the name of the output file (with suffix **".tex"**).
- otherwise the filename is **"refman.tex"**.

Parameter **output_dir** specifies the directory where to save the desired output.

Parameter **output_type** specifies the target of the operation and may be either 'TEX', 'DVI', or 'PDF'. In the latter case, method **close()** will try to start **latex** or **pdflatex** in order to produce a document of the demanded kind. If this production fails, at least the produced LaTeX source is copied to **output_dir**. The programs **latex** or **pdflatex** resp. are assumed to be in the path of executable programs.

4.3 Function close

Synopsis: **close(***self***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

Write closing statements to the LaTeX-document and **close** file. Produce DVI and PDF is requested by **m_output_type**.

4.4 Function headline1

Synopsis: **headline1(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.5 Function headline2

Synopsis: **headline2(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.6 Function headline3

Synopsis: **headline3(***self, context, title***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.7 Function write

Synopsis: **write(***self, context, standard_text, underscoreIsBlank=False, quotState='left'***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.8 Function write_class_header

Synopsis: **write_class_header(***self, modname, classname, nom, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.9 Function write_class_synopsis

Synopsis: **write_class_synopsis(***self, context, listofmodules, base_classes, extending_classes, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.10 Function write_codelinebreak

Synopsis: **write_codelinebreak(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.11 Function write_codelineinsertion

Synopsis: **write_codelineinsertion(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.12 Function write_fn_synopsis

Synopsis: **write_fn_synopsis(***self, context, argspec, base_implementations, related_implementations, unrelated_implementations, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.13 Function write_function_header

Synopsis: **write_function_header(***self, context, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.14 Function write_header_fn_section

Synopsis: **write_header_fn_section(***self, modname***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.15 Function write_index

Synopsis: **write_index(***self, context, indexentries, mainIndex=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.16 Function write_item

Synopsis: **write_item(***self, context, modeidentifier, itemname=None***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.17 Function write_label

Synopsis: **write_label(***self, context, labelname***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.18 Function write_linebreak

Synopsis: **write_linebreak(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.19 Function write_metainfo

Synopsis: **write_metainfo(***self, object***)**

Transfers meta information as represented by class **MetaInfo** into the appropriate LaTeX macros.

4.20 Function write_mode

Synopsis: **write_mode(***self, context, mode, close=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.21 Function write_multi_module_header

Synopsis: **write_multi_module_header(***self, modname, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.22 Function write_ref

Synopsis: **write_ref(***self, context, labelname, targetContext=None***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.23 Function write_refToContext

Synopsis: **write_refToContext(***self, context, refTo, refToIsMembervar=False***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.24 Function write_single_module_header

Synopsis: **write_single_module_header(***self, modname, metainfo***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.25 Function write_undocumented

Synopsis: **write_undocumented(***self, context***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.26 Function write_vars

Synopsis: **write_vars(***self, context, vars, nom***)**

Base implementations in classes: Formatter

Related implementations in classes: FormatterHTB

4.27 Member FormatterLaTeX

m_output_type = 'TEX'

m_german_preamble =

```
'''

 \usepackage{german}
 \renewcommand{\basesname}{Oberklassen:}
 \renewcommand{\extensionsname}{Unterklassen:}
 \renewcommand{\reftolabel}[1]{ \ref{#1} (vgl. Seite~\pageref{#1})}
 \renewcommand{\undocumentedelement}{\par {\em Keine Dokumentation vorhanden.}\/}
 \renewcommand{\modulename}{Modul}
 \renewcommand{\classname}{Klasse}
 \renewcommand{\methodname}{Methode}
 \renewcommand{\membername}{Merkmale}
 \renewcommand{\globalsname}{Globale Variablen}
 \renewcommand{\functionname}{Funktion}
 \renewcommand{\functionsname}{Funktionen}
 \renewcommand{\baseimplementations}{{\"U}berl{\"a}dt Implementationen in den
                folgenden Klassen:}
 \renewcommand{\relatedimplementations}{Weitere gleichnamige Methoden innerhalb
                der Vererbungshierarchie in folgenden Klassen:}
 \renewcommand{\unrelatedimplementations}{Weitere gleichnamige Methoden
                au{\"s}erhalb der Verebungshierarchie in folgenden Klassen:}
 \renewcommand{\footingmark}{Erstellt durch  {\em pymanual} (c) 2005-2006 am \today{}.}

'''
```

m_output_dir = '.'

m_tempdir = None

m_filename = None

print_verbatim = False

5 Class FormattingContext

This class is simply a triple providing the module **object**, the class **object**, and the member **object** a certain documentation is related to. Class and member information may be none (in which case the documentation is directly related to the module). This context is for some **Formatter** necessary for instance iff they produce different files.

The construction of this class (cf. **__init__()**) implements the conversion from elements to contexts.

5.1 Function __eq__

Synopsis: **__eq__(***self, obj***)**

Comparison via **moduleName()**, **className()**, and **memberName()**.

5.2 Function __ge__

Synopsis: **__ge__(***self, obj***)**

5.3 Function __gt__

Synopsis: **__gt__(***self, obj***)**

5.4 Function __hash__

Synopsis: **__hash__(***self***)**

Returns hash value for use in the context of dictionaries.

5.5 Function __init__

Synopsis: **__init__(***self, module_info=None, class_info=None, member_info=None***)**

Initializing the member. module_info should always be given except on translating strings affecting the whole document. module_info might be a class, a function, or a method. In these cases, **m_module** is set to the module containing element module_info, **m_class** is set to the name of the class containing the element if appropriate, and **m_member** is set to the element's name if appropriate..

5.6 Function __le__

Synopsis: **__le__(***self, obj***)**

5.7 Function __lt__

Synopsis: **__lt__(***self, obj***)**

5.8 Function __ne__

Synopsis: **__ne__(***self, obj***)**

5.9 Function __repr__

Synopsis: **__repr__(***self***)**

This is in contrast to **__str__()** the print out of a constructor call generating an equivalent instance.

5.10 Function __str__

Synopsis: **__str__(***self***)**

5.11 Function className

Synopsis: **className(***self***)**

Return the name of the class in the context or **None** if this context is not within a class.

5.12 Function classObject

Synopsis: **classObject(***self***)**

Return the class **object** that is represented in this context. This is either **m_class** or the class of this name in the module **object**.

5.13 Function fnObject

Synopsis: **fnObject(***self***)**

Return the function **object** that is represented by this context. This is either **m_member** or the **object** in the class **object** from the module **object** or **None**.

5.14 Function get_label

Synopsis: **get_label(***self***)**

A context is a unique identification of a particular Python code element like modules, classes, functions, or methods. This method produces a string for identifying this element that as used throughout the documentation as label for hyper targets. Since this format is relevant for using the resulting documents as online documentation, here are the relevant code fragments:

Labels for modules:

```
"module_" +only_letters_digits( self.moduleName() )
```

Labels for classes:

```
"class_"+only_letters_digits( self.className() )
    +"_"+only_letters_digits( self.moduleName() )
```

Labels for methods:

```
"function_"+only_letters_digits( self.moduleName() )
           +"_"+only_letters_digits( self.className() )
           +"_"+only_letters_digits( self.memberName() )
```

Labels for global functions:

```
"function_"+only_letters_digits(self.moduleName())
       +"_"+only_letters_digits(self.memberName())
```

5.15 Function get_level

Synopsis: **get_level(***self***)**

Return an integer number representing insertion level in a table of contents. Modules have insertion level 0. Classes and global functions have insertion level 1. Methods have insertion level 2.

5.16 Function get_name

Synopsis: **get_name(***self, context=None***)**

Returns a string that can be used as a name description in formatted output. In detail: If context refers to a different module than print **moduleName()** as qualifier. In any case print **className()** if this exists and finally **memberName()** if existing. Put a dot between printed parts of the context.

5.17 Function isClass

Synopsis: **isClass(***self***)**

True if this describes a class.

5.18 Function isFunction

Synopsis: **isFunction(***self***)**

True iff this describes a function on a module.

5.19 Function isMethod

Synopsis: **isMethod(***self***)**

True iff this describes a method of a class.

5.20 Function isModule

Synopsis: **isModule(***self***)**

True iff this describes a module.

5.21 Function isNeighbour

Synopsis: **isNeighbour(***self, context***)**

This returns **True** iff this context differs only from argument context in the most specific descriptor. This implies that both contexts are not equal to each other. Two contexts denoting different modules are neighbors. Two contexts denoting different classes or global functions in the same module are neighbours. Two different methods of the same class are neighbours.

5.22 Function isPartOf

Synopsis: **isPartOf(***self, argContext***)**

Returns **True** if **argContext** is a module and **self** describes a class or a function of this module or **argContext** is a class and **self** describes a method of this class.

5.23 Function memberName

Synopsis: **memberName(***self***)**

Return the name of the class member in the context or **None** if this context is not within a class member.

5.24 Function moduleName

Synopsis: **moduleName(***self***)**

Return the name of the module in the context.

5.25 Function moduleObject

Synopsis: **moduleObject(***self***)**

Return the module **object** that is represented in this context. This is either **m_module** or the class of this name in the module **object**.

5.26 Function object

Synopsis: **object(***self***)**

Return the represented **object**.

5.27 Function resetClassInfo

Synopsis: **resetClassInfo(***self***)**

Resetting information class and member.

5.28 Function resetMemberInfo

Synopsis: **resetMemberInfo(***self***)**

Resetting the **m_member**.

5.29 Function setClassInfo

Synopsis: **setClassInfo(***self, class_info***)**

Resetting information on member and class and setting the class information.

5.30 Function setMemberInfo

Synopsis: **setMemberInfo(***self, member_info***)**

Setting the information on the member.

5.31 Member FormattingContext

m_module = None

m_member = None

m_class = None

6 Class MetaInfo

Instances of this class represent the meta information on code elements. Currently, elements may contain the following meta attributes:

__author__ or __authors__: a string or a list of optionally multi-line strings describing the author or the authors of the element.

__copyright__: a string describing copyright information.

__version__: an instance (mostly a floating point number) describing the version of the element.

Depending on the used formatter, these elements will be presented in the documentation if present. Missing meta info will be inherited from the module or class if this is possible.

Instance creation implements collecting and inheritance of meta information. Members of name with suffix “Inherited” hold an inherited value. So, original changes may be recognized by comparison with the inherited value.

m_authorObjectName holds the name of the object describing the author in m_author or ” if there is no author. **m_versionObjectName** and **m_copyrightObjectName** analogously.

6.1 Function __init__

Synopsis: **__init__(***self, element***)**

Collects and possibly inherits meta information from the provided code element.

6.2 Function authorDescr

Synopsis: **authorDescr(***self***)**

Returns a string **m_authorObjectName** + ’ ’ + **m_authors** or ” if not specified.

6.3 Function copyrightDescr

Synopsis: **copyrightDescr(***self***)**

Returns **m_copyrightObjectName** + ’ ’ + **m_copyright** or ” if not specified.

6.4 Function versionDescr

Synopsis: **versionDescr(***self***)**

Returns **m_versionObjectName** + ' ' + **m_version** or '' if not specified.

6.5 Member MetaInfo

m_authorObjectName = ''

m_version = None

m_versionInherited = None

m_versionObjectName = ''

m_copyrightObjectName = ''

m_authors = []

m_authorsInherited = []

m_copyright = ''

m_copyrightInherited = []

7 Class doc_printer

Generate an instance of this class to produce the documentation of a single element of the implementation, either a module, a class, or a function/method.

Instances of this class are generated to print a __**doc**__ string. The work is already done during construction. This function needs to be implemented by a class since we have to keep track of a state describing whether to use "left" or "right" quotes (string value of **m_quotState**).

m_linemode is a stack describing the currently valid modes. Modes usually correspond with open LaTeX environments that have to be closed after finishing a documentation. Mode **[]** denotes normal lines. Valid modes with a single entry are **["code"]**, and **["verbatim"]** for normal text, or code lines as they are used by module **doctest**, or verbatim output repectively. The list may have more than one entry on item lists (keyword **"itemize"**) or enumeration lists (keyword "enumerate").

Instances of this class print into a specialization of **Formatter**.

7.1 Function __init__

Synopsis: **__init__(***self, dest, listofmodules, context, doc_string***)**

Refer to **print_doc**.

dest: A specialization of class **Formatter** as destination for output.

listofmodule: A list of all names of modules to be documented.

context: An instance of **FormattingContext** representing the object to be documented.

doc string: The plain documentation of the object.

7.2 Function add_mode

Synopsis: **add_mode(***self, dest, context, mode_identifier***)**

This function simply adds a mode to the stack. However, it handles only those modes where simply adding to a current state is useful. These are currently the modes “code” and “verbatim”.

context: is an instance of class **FormattingContext** describing the current context of the documentation.

dest: A specialization of class **Formatter** as destination for output.

7.3 Function adjust_mode

Synopsis: **adjust_mode(***self, dest, context, mode_identifier***)**

This method adjusts **m_linemode** to a token representing insertion and itemization in item lists or enumeration lists. If **mode_identifier** is completely composed of the chars **:**, <*>, or **#**, then this string defines a new mode with referenze to the current mode. **:** is used to inherit the old mode. <*> either sets a new item or, if the old mode is not an item list, closes mode until this level and starts a new item list. **#** does the same on enumeration lists.

Please note that mode identifiers are only recognized at the beginning of a new line. Further, modes for emphasize, ***code extracts****, and all others are cleared on starting a new line in the base text.*

Parameter context is an instance of class **FormattingContext** describing the current context of the documentation.

Some samples: (Please note, that this line would begin a description list if starting with a single word before the colon.)

- The current mode is **[“itemize”]**:

  1. **mode_identifier** “::” tells this method to do nothing. So, the text of this line will be appended to the last item of the enumerate list.
  2. **mode_identifier** “:#” tells this method to start a new item in the inner enumeration list.
  3. **mode_identifier** “:*“ tells this method to close the current item of the enumeration list and start a new item list instead.
  4. **mode_identifier** “:” tells this method to close the current item of the enumeration list and continue with the last item of the outer item list.
  5. **mode_identifier** “*“ starts a new item of the outer item list. The inner enumeration gets closed.
  6. **mode_identifier** “#” starts a new enumeration. Both open lists get closed.

  description lists: This is an item in a description list within the first item of the item list.

  on items in lists: Description lists are started on descripton items that are specified by a string (no blanks, no colons, no asterisk, and no double cross ’#’) followed by a colon. Use the underscore to replace blanks in items of descrition lists.

A verbatim excerpt from the code producing the list above:

```
* The current mode is ["itemize"]:
:# <mode_identifier> "::" tells this method to do nothing. So, the text of
:: this line will be appended to the last item of the enumerate list.
:# <mode_identifier> ":#" tells this method to start a new item in the inner
...
:description_lists: This is an item in a description list within the first item of
:: the item list.
:on_items_in_lists: Description lists are started on descripton items that
...
```

The functions returns a **True** iff **mode_identifier** defines a mode.

Type setting of lines compatible to module **doctest**:

```
>>> 3+4   # an expression to be evaluated by <doctest>
        7
>>> 7-3   # another expression to be evaluated by <doctest>
4
```

Type setting of lines in python code with highlight but irrelevant to **doctest**.

```
|>> print FormattingContext(''module'', ''class'', ''method'').get_label()
```

This should be set as a normal line since we loose any formatting directive by the empty line above in the unformatted text.

7.4 Function close_modes

Synopsis: **close_modes(***self, dest, context, modes_to_close***)**

This function closes the modes in modes_to_close from element 0 to the last element.

context: is an instance of class **FormattingContext** describing the current context of the documentation.

dest: A specialization of class **Formatter** as destination for output.

7.5 Function get_context_of_class

Synopsis: **get_context_of_class(***self, listofmodules, class_object***)**

Returns a **FormattingContext** describing the class as represented by **class_object**. This is usually **FormattingContext(class_object.__module__, class_object.__name__)**. However, if **class_object.__module__** is not in **listofmodules**, **listofmodules** is searched for a module providing this class. This is because **class_object** may have been imported directly into another module that is to be documented.

7.6 Function new_mode

Synopsis: **new_mode(***self, dest, context, new_mode***)**

This switches **m_linemode**. The method also closes open modes that are no longer needed. Therefore, the destination **Formatter** **dest** has to be known. This method can be used to generate modes “”, “code” and “verbatim”. “” means **m_linemode** **[]**.

context: is an instance of class **FormattingContext** describing the current context of the documentation.

dest: A specialization of class **Formatter** as destination for output.

7.7 Function print_class_indices

Synopsis: **print_class_indices(***self, dest, context, class_context, manyModules***)**

Print indices for the class as given in class_context into context. dest is the output formatter and manyModules is **True** iff this documentation is about more than one module.

7.8 Function print_class_synopsis

Synopsis: **print_class_synopsis(***self, dest, listofmodules, context***)**

Prints text describing base classes and specicializations if there are any. Searches listofmodules for specializations. Prints on **dest** which is supposed to be a **Formatter**. Parameter **context** describes the class to be documented.

7.9 Function print_doc

Synopsis: **print_doc(***self, dest, listofmodules, context, doc_string***)**

Helping function for **pymanual**. Format doc_string and write it to **dest**, a specialization of class **Formatter**.

listofmodules: The list of the names of all modules to eb documented.

context: A **FormattingContext** describing the object to be documented

7.10 Function print_fn_indices

Synopsis: **print_fn_indices(***self, dest, context, fn_context, manyModules***)**

Print all the index tags referring to the function as described by fn_context. context describes where the index is assumed to point at. This index entry is considered to be the main entry iff context equals fn_context.

7.11 Function print_fn_synopsis

Synopsis: **print_fn_synopsis(***self, dest, listofmodules, context***)**

Print the synopsis of a function element as described by context.

7.12 Member doc_printer

m_linemode = []

m_quotState = 'left'

8 Functions pymanual

8.1 Function only_letters_digits

Synopsis: **only_letters_digits(***str***)**

Return the string argument **str** but ignore any character in **str** that is neither a letter nor a number. This is used to generate labels (refer to **get_label()**).

8.2 Function pymanual_htb

Synopsis: **pymanual_htb(***listofmodules, output_dir, output_type, pattern, texts, title, additionalFiles***)**

Translate all __**doc**__ strings of elements of the modules with names from **listofmodules**. **listofmodules** might also be a string of a single module to be described.

Generates a **FormatterHTB** (refer to **FormatterHTB.__init__()** for a description of the arguments) and starts **pymanual_toFormatter**.

8.3 Function pymanual_latex

Synopsis: **pymanual_latex(***listofmodules, output_dir, output_type, preamble=''***)**

Translate all __**doc**__ strings of elements of the modules with names from **listofmodules**. **listofmodules** might also be a string of a single module to be described.

output_dir is the destination director. **output_type** may be either "TEX", "DVI", or "PDF" for producing a document of the desired type.

preamble is an optional extension of the standard LaTeX **preamble** to incorporate for instance translations of the builtin strings into another language.

This is the central function to produce documentations in LaTeX format.

8.4 Function pymanual_toFormatter

Synopsis: **pymanual_toFormatter(***listofmodules, dest***)**

Translate all __**doc**__ strings of elements of the modules with names from **listofmodules**. **listofmodules** might also be a string of a single module to be described.

dest is the destination of the documentation, an instance of **Formatter**.

This is the central function to produce documentations in any format supported by a formatter.

8.5 Function split_words

Synopsis: **split_words(***words, sep, exceptions=[]***)**

Argument **words** is a list of strings. Each of these lists will be splitted by separator **sep**. The resulting lists will be concatenated where **sep** is inserted where it has caused a split. This list is the result. Argument **exceptions** is an optional list of **words** that will no be splitted.

Index

