
Documenting Python

Release 2.1.2

Fred L. Drake, Jr.

January 16, 2002

PythonLabs

E-mail: fdrake@acm.org

Abstract

The Python language has a substantial body of documentation, much of it contributed by various authors. The markup used for the Python documentation is based on \LaTeX and requires a significant set of macros written specifically for documenting Python. This document describes the macros introduced to support Python documentation and how they should be used to support a wide range of output formats.

This document describes the document classes and special markup used in the Python documentation. Authors may use this guide, in conjunction with the template files provided with the distribution, to create or maintain whole documents or sections.

Contents

1	Introduction	2
2	Directory Structure	2
3	\LaTeX Primer	3
3.1	Syntax	4
3.2	Hierarchical Structure	6
4	Document Classes	6
5	Special Markup Constructs	6
5.1	Markup for the Preamble	7
5.2	Meta-information Markup	7
5.3	Information Units	7
5.4	Showing Code Examples	8
5.5	Inline Markup	9
5.6	Module-specific Markup	12
5.7	Library-level Markup	13
5.8	Table Markup	13
5.9	Reference List Markup	14
5.10	Index-generating Markup	15
6	Special Names	16
7	Processing Tools	16
7.1	External Tools	16
7.2	Internal Tools	17

8 Future Directions	17
8.1 Structured Documentation	18
8.2 Discussion Forums	18

1 Introduction

Python’s documentation has long been considered to be good for a free programming language. There are a number of reasons for this, the most important being the early commitment of Python’s creator, Guido van Rossum, to providing documentation on the language and its libraries, and the continuing involvement of the user community in providing assistance for creating and maintaining documentation.

The involvement of the community takes many forms, from authoring to bug reports to just plain complaining when the documentation could be more complete or easier to use. All of these forms of input from the community have proved useful during the time I’ve been involved in maintaining the documentation.

This document is aimed at authors and potential authors of documentation for Python. More specifically, it is for people contributing to the standard documentation and developing additional documents using the same tools as the standard documents. This guide will be less useful for authors using the Python documentation tools for topics other than Python, and less useful still for authors not using the tools at all.

The material in this guide is intended to assist authors using the Python documentation tools. It includes information on the source distribution of the standard documentation, a discussion of the document types, reference material on the markup defined in the document classes, a list of the external tools needed for processing documents, and reference material on the tools provided with the documentation resources. At the end, there is also a section discussing future directions for the Python documentation and where to turn for more information.

2 Directory Structure

The source distribution for the standard Python documentation contains a large number of directories. While third-party documents do not need to be placed into this structure or need to be placed within a similar structure, it can be helpful to know where to look for examples and tools when developing new documents using the Python documentation tools. This section describes this directory structure.

The documentation sources are usually placed within the Python source distribution as the top-level directory ‘Doc/’, but are not dependent on the Python source distribution in any way.

The ‘Doc/’ directory contains a few files and several subdirectories. The files are mostly self-explanatory, including a ‘README’ and a ‘Makefile’. The directories fall into three categories:

Document Sources

The \LaTeX sources for each document are placed in a separate directory. These directories are given short names which vaguely indicate the document in each:

Directory	Document Title
api/	<i>The Python/C API</i>
dist/	<i>Distributing Python Modules</i>
doc/	<i>Documenting Python</i>
ext/	<i>Extending and Embedding the Python Interpreter</i>
inst/	<i>Installing Python Modules</i>
lib/	<i>Python Library Reference</i>
mac/	<i>Macintosh Module Reference</i>
ref/	<i>Python Reference Manual</i>
tut/	<i>Python Tutorial</i>

Format-Specific Output

Most output formats have a directory which contains a ‘Makefile’ which controls the generation of that format and provides storage for the formatted documents. The only variations within this category are the Portable Document Format (PDF) and PostScript versions are placed in the directories ‘paper-a4/’ and ‘paper-letter/’ (this causes all the temporary files created by L^AT_EX to be kept in the same place for each paper size, where they can be more easily ignored).

Directory	Output Formats
html/	HTML output
info/	GNU info output
paper-a4/	PDF and PostScript, A4 paper
paper-letter/	PDF and PostScript, US-Letter paper

Supplemental Files

Some additional directories are used to store supplemental files used for the various processes. Directories are included for the shared L^AT_EX document classes, the L^AT_EX2HTML support, template files for various document components, and the scripts used to perform various steps in the formatting processes.

Directory	Contents
perl/	Support for L ^A T _E X2HTML processing
templates/	Example files for source documents
texinputs/	Style implementation for L ^A T _E X
tools/	Custom processing scripts

3 L^AT_EX Primer

This section is a brief introduction to L^AT_EX concepts and syntax, to provide authors enough information to author documents productively without having to become “T_EXnicians.”

Perhaps the most important concept to keep in mind while marking up Python documentation is that while T_EX is unstructured, L^AT_EX was designed as a layer on top of T_EX which specifically supports structured markup. The Python-specific markup is intended to extend the structure provided by standard L^AT_EX document classes to support additional information specific to Python.

L^AT_EX documents contain two parts: the preamble and the body. The preamble is used to specify certain metadata about the document itself, such as the title, the list of authors, the date, and the *class* the document belongs to. Additional information used to control index generation and the use of bibliographic databases can also be placed in the preamble. For most authors, the preamble can be most easily created by copying it from an existing document and modifying a few key pieces of information.

The *class* of a document is used to place a document within a broad category of documents and set some fundamental formatting properties. For Python documentation, two classes are used: the `manual` class and the `howto` class. These classes also define the additional markup used to document Python concepts and structures. Specific information about

these classes is provided in section 4, “Document Classes,” below. The first thing in the preamble is the declaration of the document’s class.

After the class declaration, a number of *macros* are used to provide further information about the document and setup any additional markup that is needed. No output is generated from the preamble; it is an error to include free text in the preamble because it would cause output.

The document body follows the preamble. This contains all the printed components of the document marked up structurally. Generic L^AT_EX structures include hierarchical sections, numbered and bulleted lists, and special structures for the document abstract and indexes.

3.1 Syntax

There are some things that an author of Python documentation needs to know about L^AT_EX syntax.

A *comment* is started by the “percent” character (%) and continues through the end of the line and all leading whitespace on the following line. This is a little different from any programming language I know of, so an example is in order:

```
This is text.% comment
    This is more text. % another comment
Still more text.
```

The first non-comment character following the first comment is the letter ‘T’ on the second line; the leading whitespace on that line is consumed as part of the first comment. This means that there is no space between the first and second sentences, so the period and letter ‘T’ will be directly adjacent in the typeset document.

Note also that though the first non-comment character after the second comment is the letter ‘S’, there is whitespace preceding the comment, so the two sentences are separated as expected.

A *group* is an enclosure for a collection of text and commands which encloses the formatting context and constrains the scope of any changes to that context made by commands within the group. Groups can be nested hierarchically. The formatting context includes the font and the definition of additional macros (or overrides of macros defined in outer groups). Syntactically, groups are enclosed in braces:

```
{text in a group}
```

An alternate syntax for a group using brackets, [. . .], is used by macros and environment constructors which take optional parameters; brackets do not normally hold syntactic significance. A degenerate group, containing only one atomic bit of content, does not need to have an explicit group, unless it is required to avoid ambiguity. Since Python tends toward the explicit, groups are also made explicit in the documentation markup.

Groups are used only sparingly in the Python documentation, except for their use in marking parameters to macros and environments.

A *macro* is usually a simple construct which is identified by name and can take some number of parameters. In normal L^AT_EX usage, one of these can be optional. The markup is introduced using the backslash character (‘\’), and the name is given by alphabetic characters (no digits, hyphens, or underscores). Required parameters should be marked as a group, and optional parameters should be marked using the alternate syntax for a group.

For example, a macro named “foo” which takes a single parameter would appear like this:

```
\name{parameter}
```

A macro which takes an optional parameter would be typed like this when the optional parameter is given:

```
\name[optional]
```

If both optional and required parameters are to be required, it looks like this:

```
\name[optional]{required}
```

A macro name may be followed by a space or newline; a space between the macro name and any parameters will be consumed, but this usage is not practiced in the Python documentation. Such a space is still consumed if there are no parameters to the macro, in which case inserting an empty group (`{ }`) or explicit word space (`'\ '`) immediately after the macro name helps to avoid running the expansion of the macro into the following text. Macros which take no parameters but which should not be followed by a word space do not need special treatment if the following character in the document source is not a name character (such as punctuation).

Each line of this example shows an appropriate way to write text which includes a macro which takes no parameters:

```
This \UNIX{} is followed by a space.  
This \UNIX\ is also followed by a space.  
\UNIX, followed by a comma, needs no additional markup.
```

An *environment* is a larger construct than a macro, and can be used for things with more content than would conveniently fit in a macro parameter. They are primarily used when formatting parameters need to be changed before and after a large chunk of content, but the content itself needs to be highly flexible. Code samples are presented using an environment, and descriptions of functions, methods, and classes are also marked using environments.

Since the content of an environment is free-form and can consist of several paragraphs, they are actually marked using a pair of macros: `\begin` and `\end`. These macros both take the name of the environment as a parameter. An example is the environment used to mark the abstract of a document:

```
\begin{abstract}  
  This is the text of the abstract.  It concisely explains what  
  information is found in the document.  
  
  It can consist of multiple paragraphs.  
\end{abstract}
```

An environment can also have required and optional parameters of its own. These follow the parameter of the `\begin` macro. This example shows an environment which takes a single required parameter:

```
\begin{datadesc}{controlnames}  
  A 33-element string array that contains the \ASCII{} mnemonics for  
  the thirty-two \ASCII{} control characters from 0 (NUL) to 0x1f  
  (US), in order, plus the mnemonic \samp{SP} for the space character.  
\end{datadesc}
```

There are a number of less-used marks in \LaTeX which are used to enter non-ASCII characters, especially those used in European names. Given that these are often used adjacent to other characters, the markup required to produce the proper character may need to be followed by a space or an empty group, or the markup can be enclosed in a group. Some which are found in Python documentation are:

Character	Markup
ç	<code>\c c</code>
ö	<code>\"o</code>
ø	<code>\o</code>

3.2 Hierarchical Structure

\LaTeX expects documents to be arranged in a conventional, hierarchical way, with chapters, sections, sub-sections, appendixes, and the like. These are marked using macros rather than environments, probably because the end of a section can be safely inferred when a section of equal or higher level starts.

There are six “levels” of sectioning in the document classes used for Python documentation, and the deepest two levels¹ are not used. The levels are:

Level	Macro Name	Notes
1	<code>\chapter</code>	(1)
2	<code>\section</code>	
3	<code>\subsection</code>	
4	<code>\subsubsection</code>	
5	<code>\paragraph</code>	(2)
6	<code>\subparagraph</code>	

Notes:

(1) Only used for the `manual` documents, as described in section 4, “Document Classes.”

(2) Not the same as a paragraph of text; nobody seems to use this.

4 Document Classes

Two \LaTeX document classes are defined specifically for use with the Python documentation. The `manual` class is for large documents which are sectioned into chapters, and the `howto` class is for smaller documents.

The `manual` documents are larger and are used for most of the standard documents. This document class is based on the standard \LaTeX `report` class and is formatted very much like a long technical report. The *Python Reference Manual* is a good example of a `manual` document, and the *Python Library Reference* is a large example.

The `howto` documents are shorter, and don’t have the large structure of the `manual` documents. This class is based on the standard \LaTeX `article` class and is formatted somewhat like the Linux Documentation Project’s “HOWTO” series as done originally using the LinuxDoc software. The original intent for the document class was that it serve a similar role as the LDP’s HOWTO series, but the applicability of the class turns out to be somewhat broader. This class is used for “how-to” documents (this document is an example) and for shorter reference manuals for small, fairly cohesive module libraries. Examples of the later use include the standard *Macintosh Library Modules* and *Using Kerberos from Python*, which contains reference material for an extension package. These documents are roughly equivalent to a single chapter from a larger work.

5 Special Markup Constructs

The Python document classes define a lot of new environments and macros. This section contains the reference material for these facilities.

¹The deepest levels have the highest numbers in the table.

5.1 Markup for the Preamble

`\release{ver}`

Set the version number for the software described in the document.

`\setshortversion{sver}`

Specify the “short” version number of the documented software to be *sver*.

5.2 Meta-information Markup

`\sectionauthor{author}{email}`

Identifies the author of the current section. *author* should be the author’s name such that it can be used for presentation (though it isn’t), and *email* should be the author’s email address. The domain name portion of the address should be lower case.

No presentation is generated from this markup, but it is used to help keep track of contributions.

5.3 Information Units

XXX Explain terminology, or come up with something more “lay.”

There are a number of environments used to describe specific features provided by modules. Each environment requires parameters needed to provide basic information about what is being described, and the environment content should be the description. Most of these environments make entries in the general index (if one is being produced for the document); if no index entry is desired, non-indexing variants are available for many of these environments. The environments have names of the form *featuredesc*, and the non-indexing variants are named *featuredescni*. The available variants are explicitly included in the list below.

For each of these environments, the first parameter, *name*, provides the name by which the feature is accessed.

Environments which describe features of objects within a module, such as object methods or data attributes, allow an optional *type name* parameter. When the feature is an attribute of class instances, *type name* only needs to be given if the class was not the most recently described class in the module; the *name* value from the most recent *classdesc* is implied. For features of built-in or extension types, the *type name* value should always be provided. Another special case includes methods and members of general “protocols,” such as the formatter and writer protocols described for the *formatter* module: these may be documented without any specific implementation classes, and will always require the *type name* parameter to be provided.

`\begin{cfuncdesc}{type}{name}{args}`

`\end{cfuncdesc}`

Environment used to described a C function. The *type* should be specified as a `typedef` name, `struct tag`, or the name of a primitive type. If it is a pointer type, the trailing asterisk should not be preceded by a space. *name* should be the name of the function (or function-like pre-processor macro), and *args* should give the types and names of the parameters. The names need to be given so they may be used in the description.

`\begin{ctypedesc}[tag]{name}`

`\end{ctypedesc}`

Environment used to described a C type. The *name* parameter should be the `typedef` name. If the type is defined as a `struct` without a `typedef`, *name* should have the form `struct tag`. *name* will be added to the index unless *tag* is provided, in which case *tag* will be used instead. *tag* should not be used for a `typedef` name.

`\begin{cvardesc}{type}{name}`

`\end{cvardesc}`

Description of a global C variable. *type* should be the `typedef` name, `struct tag`, or the name of a primitive type. If variable has a pointer type, the trailing asterisk should *not* be preceded by a space.

```
\begin{datadesc} {name}
\end{datadesc}
```

This environment is used to document global data in a module, including both variables and values used as “defined constants.” Class and object attributes are not documented using this environment.

```
\begin{datadescni} {name}
\end{datadescni}
```

Like `datadesc`, but without creating any index entries.

```
\begin{excdesc} {name}
\end{excdesc}
```

Describe an exception. This may be either a string exception or a class exception.

```
\begin{funcdesc} {name} {parameters}
\end{funcdesc}
```

Describe a module-level function. *parameters* should not include the parentheses used in the call syntax. Object methods are not documented using this environment. Bound object methods placed in the module namespace as part of the public interface of the module are documented using this, as they are equivalent to normal functions for most purposes.

The description should include information about the parameters required and how they are used (especially whether mutable objects passed as parameters are modified), side effects, and possible exceptions. A small example may be provided.

```
\begin{funcdescni} {name} {parameters}
\end{funcdescni}
```

Like `funcdesc`, but without creating any index entries.

```
\begin{classdesc} {name} {constructor parameters}
\end{classdesc}
```

Describe a class and its constructor. *constructor parameters* should not include the *self* parameter or the parentheses used in the call syntax.

```
\begin{memberdesc} [type name] {name}
\end{memberdesc}
```

Describe an object data attribute. The description should include information about the type of the data to be expected and whether it may be changed directly.

```
\begin{memberdescni} [type name] {name}
\end{memberdescni}
```

Like `memberdesc`, but without creating any index entries.

```
\begin{methoddesc} [type name] {name} {parameters}
\end{methoddesc}
```

Describe an object method. *parameters* should not include the *self* parameter or the parentheses used in the call syntax. The description should include similar information to that described for `funcdesc`.

```
\begin{methoddescni} [type name] {name} {parameters}
\end{methoddescni}
```

Like `methoddesc`, but without creating any index entries.

5.4 Showing Code Examples

Examples of Python source code or interactive sessions are represented as `verbatim` environments. This environment is a standard part of \LaTeX . It is important to only use spaces for indentation in code examples since \TeX drops tabs instead of converting them to spaces.

Representing an interactive session requires including the prompts and output along with the Python code. No special markup is required for interactive sessions. After the last line of input or output presented, there should not be an

“unused” primary prompt; this is an example of what *not* to do:

```
>>> 1 + 1
2
>>>
```

Within the `verbatim` environment, characters special to \LaTeX do not need to be specially marked in any way. The entire example will be presented in a monospaced font; no attempt at “pretty-printing” is made, as the environment must work for non-Python code and non-code displays. There should be no blank lines at the top or bottom of any `verbatim` display.

The Python Documentation Special Interest Group has discussed a number of approaches to creating pretty-printed code displays and interactive sessions; see the Doc-SIG area on the Python Web site for more information on this topic.

5.5 Inline Markup

The macros described in this section are used to mark just about anything interesting in the document text. They may be used in headings (though anything involving hyperlinks should be avoided there) as well as in the body text.

`\bfcodes{text}`

Like `\code`, but also makes the font bold-face.

`\cdata{name}`

The name of a C-language variable.

`\cfunction{name}`

The name of a C-language function. *name* should include the function name and the trailing parentheses.

`\character{char}`

A character when discussing the character rather than a one-byte string value. The character will be typeset as with `\samp`.

`\citetitle[url]{title}`

A title for a referenced publication. If *url* is specified, the title will be made into a hyperlink when formatted as HTML.

`\class{name}`

A class name; a dotted name may be used.

`\code{text}`

A short code fragment or literal constant value. Typically, it should not include any spaces since no quotation marks are added.

`\constant{name}`

The name of a “defined” constant. This may be a C-language `#define` or a Python variable that is not intended to be changed.

`\ctype{name}`

The name of a C typedef or structure. For structures defined without a typedef, use `\ctype{struct struct_tag}` to make it clear that the `struct` is required.

`\deprecated{version}{what to do}`

Declare whatever is being described as being deprecated starting with release *version*. The text given as *what to do* should recommend something to use instead.

`\dfn{term}`

Mark the defining instance of *term* in the text. (No index entries are generated.)

- `\e`
 Produces a backslash. This is convenient in `\code` and similar macros, and is only defined there. To create a backslash in ordinary text (such as the contents of the `\file` macro), use the standard `\textbackslash` macro.
- `\email{address}`
 An email address. Note that this is *not* hyperlinked in any of the possible output formats. The domain name portion of the address should be lower case.
- `\emph{text}`
 Emphasized text; this will be presented in an italic font.
- `\envvar{name}`
 An environment variable. Index entries are generated.
- `\exception{name}`
 The name of an exception. A dotted name may be used.
- `\file{file or dir}`
 The name of a file or directory. In the PDF and PostScript outputs, single quotes and a font change are used to indicate the file name, but no quotes are used in the HTML output. **Warning:** The `\file` macro cannot be used in the content of a section title due to processing limitations.
- `\filenq{file or dir}`
 Like `\file`, but single quotes are never used. This can be used in conjunction with tables if a column will only contain file or directory names. **Warning:** The `\filenq` macro cannot be used in the content of a section title due to processing limitations.
- `\function{name}`
 The name of a Python function; dotted names may be used.
- `\kbd{key sequence}`
 Mark a sequence of keystrokes. What form *key sequence* takes may depend on platform- or application-specific conventions. For example, an **xemacs** key sequence may be marked like `\kbd{C-x C-f}`.
- `\keyword{name}`
 The name of a keyword in a programming language.
- `\makevar{name}`
 The name of a **make** variable.
- `\manpage{name}{section}`
 A reference to a UNIX manual page.
- `\member{name}`
 The name of a data attribute of an object.
- `\method{name}`
 The name of a method of an object. *name* should include the method name and the trailing parentheses. A dotted name may be used.
- `\mimetype{name}`
 The name of a MIME type.
- `\module{name}`
 The name of a module; a dotted name may be used. This should also be used for package names.
- `\newsgroup{name}`
 The name of a USENET newsgroup.
- `\program{name}`
 The name of an executable program. This may differ from the file name for the executable for some platforms. In particular, the `.exe` (or other) extension should be omitted for DOS and Windows programs.

- \programopt**{*option*}
- A command-line option to an executable program. Use this only for “shot” options, and include the leading hyphen.
- \longprogramopt**{*option*}
- A long command-line option to an executable program. This should only be used for long option names which will be prefixed by two hyphens; the hyphens should not be provided as part of *option*.
- \pep**{*number*}
- A reference to a Python Enhancement Proposal. This generates appropriate index entries. The text ‘PEP *number*’ is generated; in the HTML output, this text is a hyperlink to an online copy of the specified PEP.
- \refmodule**[*key*]{*name*}
- Like `\module`, but create a hyperlink to the documentation for the named module. Note that the corresponding `\declaremodule` must be in the same document. If the `\declaremodule` defines a module key different from the module name, it must also be provided as *key* to the `\refmodule` macro.
- \regexp**{*string*}
- Mark a regular expression.
- \rfc**{*number*}
- A reference to an Internet Request for Comments. This generates appropriate index entries. The text ‘RFC *number*’ is generated; in the HTML output, this text is a hyperlink to an online copy of the specified RFC.
- \samp**{*text*}
- A short code sample, but possibly longer than would be given using `\code`. Since quotation marks are added, spaces are acceptable.
- \shortversion**
- The “short” version number of the documented software, as specified using the `\setshortversion` macro in the preamble. For Python, the short version number for a release is the first three characters of the `sys.version` value. For example, versions 2.0b1 and 2.0.1 both have a short version of 2.0. This may not apply for all packages; if `\setshortversion` is not used, this produces an empty expansion. See also the `\version` macro.
- \strong**{*text*}
- Strongly emphasized text; this will be presented using a bold font.
- \url**{*url*}
- A URL (or URN). The URL will be presented as text. In the HTML and PDF formatted versions, the URL will also be a hyperlink. This can be used when referring to external resources. Note that many characters are special to L^AT_EX and this macro does not always do the right thing. In particular, the tilde character (“~”) is mis-handled; encoding it as a hex-sequence does work, use ‘%7e’ in place of the tilde character.
- \var**{*name*}
- The name of a variable or formal parameter in running text.
- \version**
- The version number of the described software, as specified using `\release` in the preamble. See also the `\shortversion` macro.
- \versionadded**[*explanation*]{*version*}
- The version of Python which added the described feature to the library or C API. *explanation* should be a *brief* explanation of the change consisting of a capitalized sentence fragment; a period will be appended by the formatting process. This is typically added to the end of the first paragraph of the description before any availability notes. The location should be selected so the explanation makes sense and may vary as needed.
- \versionchanged**[*explanation*]{*version*}
- The version of Python in which the named feature was changed in some way (new parameters, changed side effects, etc.). *explanation* should be a *brief* explanation of the change consisting of a capitalized sentence fragment; a period will be appended by the formatting process. This is typically added to the end of the first

paragraph of the description before any availability notes and after `\versionadded`. The location should be selected so the explanation makes sense and may vary as needed.

5.6 Module-specific Markup

The markup described in this section is used to provide information about a module being documented. A typical use of this markup appears at the top of the section used to document a module. A typical example might look like this:

```
\section{\module{spam} ---
        Access to the SPAM facility}

\declaremodule{extension}{spam}
  \platform{Unix}
\modulesynopsis{Access to the SPAM facility of \UNIX{}}
\moduleauthor{Jane Doe}{jane.doe@frobnitz.org}
```

Python packages — collections of modules that can be described as a unit — are documented using the same markup as modules. The name for a module in a package should be typed in “fully qualified” form (i.e., it should include the package name). For example, a module “foo” in package “bar” should be marked as `\module{bar.foo}`, and the beginning of the reference section would appear as:

```
\section{\module{bar.foo} ---
        Module from the \module{bar} package}

\declaremodule{extension}{bar.foo}
\modulesynopsis{Nifty module from the \module{bar} package.}
\moduleauthor{Jane Doe}{jane.doe@frobnitz.org}
```

Note that the name of a package is also marked using `\module`.

`\declaremodule`[*key*] {*type*} {*name*}

Requires two parameters: module type (`'standard'`, `'builtin'`, `'extension'`, or `'`), and the module name. An optional parameter should be given as the basis for the module’s “key” used for linking to or referencing the section. The “key” should only be given if the module’s name contains any underscores, and should be the name with the underscores stripped. Note that the *type* parameter must be one of the values listed above or an error will be printed. For modules which are contained in packages, the fully-qualified name should be given as *name* parameter. This should be the first thing after the `\section` used to introduce the module.

`\platform`{*specifier*}

Specifies the portability of the module. *specifier* is a comma-separated list of keys that specify what platforms the module is available on. The keys are short identifiers; examples that are in use include `'IRIX'`, `'Mac'`, `'Windows'`, and `'Unix'`. It is important to use a key which has already been used when applicable. This is used to provide annotations in the Module Index and the HTML and GNU info output.

`\modulesynopsis`{*text*}

The *text* is a short, “one line” description of the module that can be used as part of the chapter introduction. This is must be placed after `\declaremodule`. The synopsis is used in building the contents of the table inserted as the `\localmodule` table. No text is produced at the point of the markup.

`\moduleauthor`{*name*} {*email*}

This macro is used to encode information about who authored a module. This is currently not used to generate output, but can be used to help determine the origin of the module.

5.7 Library-level Markup

This markup is used when describing a selection of modules. For example, the *Macintosh Library Modules* document uses this to help provide an overview of the modules in the collection, and many chapters in the *Python Library Reference* use it for the same purpose.

`\localmoduleable`

If a `.syn` file exists for the current chapter (or for the entire document in `howto` documents), a `synop-sistable` is created with the contents loaded from the `.syn` file.

5.8 Table Markup

There are three general-purpose table environments defined which should be used whenever possible. These environments are defined to provide tables of specific widths and some convenience for formatting. These environments are not meant to be general replacements for the standard \LaTeX table environments, but can be used for an advantage when the documents are processed using the tools for Python documentation processing. In particular, the generated HTML looks good! There is also an advantage for the eventual conversion of the documentation to SGML (see section 8, “Future Directions”).

Each environment is named `tablecols`, where `cols` is the number of columns in the table specified in lower-case Roman numerals. Within each of these environments, an additional macro, `\linecols`, is defined, where `cols` matches the `cols` value of the corresponding table environment. These are supported for `cols` values of `ii`, `iii`, and `iv`. These environments are all built on top of the `tabular` environment. Variants based on the `longtable` environment are also provided.

Note that all tables in the standard Python documentation use vertical lines between columns, and this must be specified in the markup for each table. A general border around the outside of the table is not used, but would be the responsibility of the processor; the document markup should not include an exterior border.

The `longtable`-based variants of the table environments are formatted with extra space before and after, so should only be used on tables which are long enough that splitting over multiple pages is reasonable; tables with fewer than twenty rows should never be marked using the long flavors of the table environments. The header row is repeated across the top of each part of the table.

```
\begin{tableii}{colspec}{colfont}{heading1}{heading2}
\end{tableii}
```

Create a two-column table using the \LaTeX column specifier `colspec`. The column specifier should indicate vertical bars between columns as appropriate for the specific table, but should not specify vertical bars on the outside of the table (that is considered a stylesheet issue). The `colfont` parameter is used as a stylistic treatment of the first column of the table: the first column is presented as `\colfont{column1}`. To avoid treating the first column specially, `colfont` may be `texttrm`. The column headings are taken from the values `heading1` and `heading2`.

```
\begin{longtableii}...
\end{longtableii}
```

Like `tableii`, but produces a table which may be broken across page boundaries. The parameters are the same as for `tableii`.

```
\lineii{column1}{column2}
```

Create a single table row within a `tableii` or `longtableii` environment. The text for the first column will be generated by applying the macro named by the `colfont` value when the `tableii` was opened.

```
\begin{tableiii}{colspec}{colfont}{heading1}{heading2}{heading3}
\end{tableiii}
```

Like the `tableii` environment, but with a third column. The heading for the third column is given by `heading3`.

```
\begin{longtableiii}...
```

`\end{longtableiii}`
 Like `tableiii`, but produces a table which may be broken across page boundaries. The parameters are the same as for `tableiii`.

`\lineiii{column1}{column2}{column3}`
 Like the `\lineii` macro, but with a third column. The text for the third column is given by `column3`.

`\begin{tableiv}{colspec}{colfont}{heading1}{heading2}{heading3}{heading4}`
`\end{tableiv}`
 Like the `tableiii` environment, but with a fourth column. The heading for the fourth column is given by `heading4`.

`\begin{longtableiv}...`
`\end{longtableiv}`
 Like `tableiv`, but produces a table which may be broken across page boundaries. The parameters are the same as for `tableiv`.

`\lineiv{column1}{column2}{column3}{column4}`
 Like the `\lineiii` macro, but with a fourth column. The text for the fourth column is given by `column4`.

An additional table-like environment is `synopsistable`. The table generated by this environment contains two columns, and each row is defined by an alternate definition of `\modulesynopsis`. This environment is not normally used by authors, but is created by the `\localmoduletable` macro.

5.9 Reference List Markup

Many sections include a list of references to module documentation or external documents. These lists are created using the `seealso` environment. This environment defines some additional macros to support creating reference entries in a reasonable manner.

The `seealso` environment is typically placed in a section just before any sub-sections. This is done to ensure that reference links related to the section are not hidden in a subsection in the hypertext renditions of the documentation.

`\begin{seealso}`
`\end{seealso}`

This environment creates a “See also:” heading and defines the markup used to describe individual references.

For each of the following macros, *why* should be one or more complete sentences, starting with a capital letter (unless it starts with an identifier, which should not be modified), and ending with the appropriate punctuation.

These macros are only defined within the content of the `seealso` environment.

`\seemodule[key]{name}{why}`

Refer to another module. *why* should be a brief explanation of why the reference may be interesting. The module name is given in *name*, with the link key given in *key* if necessary. In the HTML and PDF conversions, the module name will be a hyperlink to the referred-to module. **Note:** The module must be documented in the same document (the corresponding `\declaremodule` is required).

`\seepep{number}{title}{why}`

Refer to an Python Enhancement Proposal (PEP). *number* should be the official number assigned by the PEP Editor, *title* should be the human-readable title of the PEP as found in the official copy of the document, and *why* should explain what’s interesting about the PEP. This should be used to refer the reader to PEPs which specify interfaces or language features relevant to the material in the annotated section of the documentation.

`\seerfc{number}{title}{why}`

Refer to an IETF Request for Comments (RFC). Otherwise very similar to `\seepep`. This should be used to refer the reader to PEPs which specify protocols or data formats relevant to the material in the annotated section of the documentation.

`\seetext{text}`

Add arbitrary text *text* to the “See also:” list. This can be used to refer to off-line materials or on-line materials using the `\url` macro. This should consist of one or more complete sentences.

`\seetitle`*[url]* *{title}* *{why}*

Add a reference to an external document named *title*. If *url* is given, the title is made a hyperlink in the HTML version of the documentation, and displayed below the title in the typeset versions of the documentation.

`\seeurl`*{url}* *{why}*

References to specific on-line resources should be given using the `\seeurl` macro. No title is associated with the reference, but the *why* text may include a title marked using the `\citetitle` macro.

5.10 Index-generating Markup

Effective index generation for technical documents can be very difficult, especially for someone familiar with the topic but not the creation of indexes. Much of the difficulty arises in the area of terminology: including the terms an expert would use for a concept is not sufficient. Coming up with the terms that a novice would look up is fairly difficult for an author who, typically, is an expert in the area she is writing on.

The truly difficult aspects of index generation are not areas with which the documentation tools can help. However, ease of producing the index once content decisions are made is within the scope of the tools. Markup is provided which the processing software is able to use to generate a variety of kinds of index entry with minimal effort. Additionally, many of the environments described in section 5.3, “Information Units,” will generate appropriate entries into the general and module indexes.

The following macro can be used to control the generation of index data, and should be used in the document preamble:

`\makemodindex`

This should be used in the document preamble if a “Module Index” is desired for a document containing reference material on many modules. This causes a data file `libjobname.idx` to be created from the `\declare-module` macros. This file can be processed by the `makeindex` program to generate a file which can be `\input` into the document at the desired location of the module index.

There are a number of macros that are useful for adding index entries for particular concepts, many of which are specific to programming languages or even Python.

`\bifuncindex`*{name}*

Add an index entry referring to a built-in function named *name*; parentheses should not be included after *name*.

`\exindex`*{exception}*

Add a reference to an exception named *exception*. The exception may be either string- or class-based.

`\kwindex`*{keyword}*

Add a reference to a language keyword (not a keyword parameter in a function or method call).

`\obindex`*{object type}*

Add an index entry for a built-in object type.

`\opindex`*{operator}*

Add a reference to an operator, such as ‘+’.

`\refmodindex`*[key]* *{module}*

Add an index entry for module *module*; if *module* contains an underscore, the optional parameter *key* should be provided as the same string with underscores removed. An index entry “*module* (module)” will be generated. This is intended for use with non-standard modules implemented in Python.

`\refexmodindex`*[key]* *{module}*

As for `\refmodindex`, but the index entry will be “*module* (extension module).” This is intended for use with non-standard modules not implemented in Python.

`\refbimodindex`*[key]* *{module}*

As for `\refmodindex`, but the index entry will be “*module* (built-in module).” This is intended for use with standard modules not implemented in Python.

`\refstmodindex`[*key*] {*module*}

As for `\refmodindex`, but the index entry will be “*module* (standard module).” This is intended for use with standard modules implemented in Python.

`\stindex`{*statement*}

Add an index entry for a statement type, such as `print` or `try/finally`.

XXX Need better examples of difference from `\kwindex`.

Additional macros are provided which are useful for conveniently creating general index entries which should appear at many places in the index by rotating a list of words. These are simple macros that simply use `\index` to build some number of index entries. Index entries build using these macros contain both primary and secondary text.

`\indexii`{*word1*}{*word2*}

Build two index entries. This is exactly equivalent to using `\index{word1!word2}` and `\index{word2!word1}`.

`\indexiii`{*word1*}{*word2*}{*word3*}

Build three index entries. This is exactly equivalent to using `\index{word1!word2 word3}`, `\index{word2!word3 word1}`, and `\index{word3!word1 word2}`.

`\indexiv`{*word1*}{*word2*}{*word3*}{*word4*}

Build four index entries. This is exactly equivalent to using `\index{word1!word2 word3 word4}`, `\index{word2!word3 word4 word1}`, `\index{word3!word4 word1 word2}`, and `\index{word4!word1 word2 word3}`.

6 Special Names

Many special names are used in the Python documentation, including the names of operating systems, programming languages, standards bodies, and the like. Many of these were assigned L^AT_EX macros at some point in the distant past, and these macros lived on long past their usefulness. In the current markup, these entities are not assigned any special markup, but the preferred spellings are given here to aid authors in maintaining the consistency of presentation in the Python documentation.

POSIX The name assigned to a particular group of standards. This is always uppercase.

Python The name of our favorite programming language is always capitalized.

Unicode The name of a character set and matching encoding. This is always written capitalized.

7 Processing Tools

7.1 External Tools

Many tools are needed to be able to process the Python documentation if all supported formats are required. This section lists the tools used and when each is required. Consult the ‘Doc/README’ file to see if there are specific version requirements for any of these.

dvips This program is a typical part of T_EX installations. It is used to generate PostScript from the “device independent” ‘.dvi’ files. It is needed for the conversion to PostScript.

emacs Emacs is the kitchen sink of programmers' editors, and a damn fine kitchen sink it is. It also comes with some of the processing needed to support the proper menu structures for Texinfo documents when an info conversion is desired. This is needed for the info conversion. Using **xemacs** instead of FSF **emacs** may lead to instability in the conversion, but that's because nobody seems to maintain the Emacs Texinfo code in a portable manner.

latex This is a world-class typesetter by Donald Knuth. It is used for the conversion to PostScript, and is needed for the HTML conversion as well (**L^AT_EX2HTML** requires one of the intermediate files it creates).

latex2html Probably the longest Perl script anyone ever attempted to maintain. This converts **L^AT_EX** documents to HTML documents, and does a pretty reasonable job. It is required for the conversions to HTML and GNU info.

lynx This is a text-mode Web browser which includes an HTML-to-plain text conversion. This is used to convert `howto` documents to text.

make Just about any version should work for the standard documents, but GNU **make** is required for the experimental processes in 'Doc/tools/sgmlconv/', at least while they're experimental.

makeindex This is a standard program for converting **L^AT_EX** index data to a formatted index; it should be included with all **L^AT_EX** installations. It is needed for the PDF and PostScript conversions.

makeinfo GNU **makeinfo** is used to convert Texinfo documents to GNU info files. Since Texinfo is used as an intermediate format in the info conversion, this program is needed in that conversion.

pdflatex pdf**T_EX** is a relatively new variant of **T_EX**, and is used to generate the PDF version of the manuals. It is typically installed as part of most of the large **T_EX** distributions. **pdflatex** is pdf**T_EX** using the **L^AT_EX** format.

perl Perl is required for **L^AT_EX2HTML** and one of the scripts used to post-process **L^AT_EX2HTML** output, as well as the HTML-to-Texinfo conversion. This is required for the HTML and GNU info conversions.

python Python is used for many of the scripts in the 'Doc/tools/' directory; it is required for all conversions. This shouldn't be a problem if you're interested in writing documentation for Python!

7.2 Internal Tools

This section describes the various scripts that are used to implement various stages of document processing or to orchestrate entire build sequences. Most of these tools are only useful in the context of building the standard documentation, but some are more general.

mkhowto This is the primary script used to format third-party documents. It contains all the logic needed to "get it right." The proper way to use this script is to make a symbolic link to it or run it in place; the actual script file must be stored as part of the documentation source tree, though it may be used to format documents outside the tree. Use **mkhowto --help** for a list of command line options.

mkhowto can be used for both `howto` and `manual` class documents. (For the later, be sure to get the latest version from the Python CVS repository rather than the version distributed in the 'latex-1.5.2.tgz' source archive.)

XXX Need more here.

8 Future Directions

The history of the Python documentation is full of changes, most of which have been fairly small and evolutionary. There has been a great deal of discussion about making large changes in the markup languages and tools used to process the documentation. This section deals with the nature of the changes and what appears to be the most likely path of future development.

8.1 Structured Documentation

Most of the small changes to the \LaTeX markup have been made with an eye to divorcing the markup from the presentation, making both a bit more maintainable. Over the course of 1998, a large number of changes were made with exactly this in mind; previously, changes had been made but in a less systematic manner and with more concern for not needing to update the existing content. The result has been a highly structured and semantically loaded markup language implemented in \LaTeX . With almost no basic \TeX or \LaTeX markup in use, however, the markup syntax is about the only evidence of \LaTeX in the actual document sources.

One side effect of this is that while we've been able to use standard "engines" for manipulating the documents, such as \LaTeX and $\LaTeX2HTML$, most of the actual transformations have been created specifically for Python. The \LaTeX document classes and $\LaTeX2HTML$ support are both complete implementations of the specific markup designed for these documents.

Combining highly customized markup with the somewhat esoteric systems used to process the documents leads us to ask some questions: Can we do this more easily? and, Can we do this better? After a great deal of discussion with the community, we have determined that actively pursuing modern structured documentation systems is worth some investment of time.

There appear to be two real contenders in this arena: the Standard General Markup Language (SGML), and the Extensible Markup Language (XML). Both of these standards have advantages and disadvantages, and many advantages are shared.

SGML offers advantages which may appeal most to authors, especially those using ordinary text editors. There are also additional abilities to define content models. A number of high-quality tools with demonstrated maturity is available, but most are not free; for those which are, portability issues remain a problem.

The advantages of XML include the availability of a large number of evolving tools. Unfortunately, many of the associated standards are still evolving, and the tools will have to follow along. This means that developing a robust tool set that uses more than the basic XML 1.0 recommendation is not possible in the short term. The promised availability of a wide variety of high-quality tools which support some of the most important related standards is not immediate. Many tools are likely to be free.

XXX Eventual migration to SGML/XML.

8.2 Discussion Forums

Discussion of the future of the Python documentation and related topics takes place in the Documentation Special Interest Group, or "Doc-SIG." Information on the group, including mailing list archives and subscription information, is available at <http://www.python.org/sigs/doc-sig/>. The SIG is open to all interested parties.

Comments and bug reports on the standard documents should be sent to python-docs@python.org. This may include comments about formatting, content, grammatical and spelling errors, or this document. You can also send comments on this document directly to the author at fdrake@acm.org.