Documenting Python

Release 1.5.2

Fred L. Drake, Jr.

April 30, 1999

Corporation for National Research Initiatives (CNRI)
1895 Preston White Drive, Reston, Va 20191, USA
E-mail: fdrake@acm.org

Abstract

The Python language documentation 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. Maintaining the documentation requires substantial effort, in part because selecting the correct markup to use is not always easy.

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

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, three-character names:

Directory	Document Title
api/	*The Python/C API*
doc/	*Documenting Python*
ext/	*Extending and Embedding the Python Interpreter*
lib/	*Python Library Reference*
mac/	*Macintosh Module Reference*
ref/	*Python Reference Manual*
tut/	*Python Tutorial*

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 LaTeX 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 LaTeX document classes, the LaTeX2HTML support, template files for various document components, and the scripts used to perform various steps in the formatting processes.

Directory	Contents
perl/	Support for LaTeX2HTML processing
templates/	Example files for source documents
texinputs/	Style implementation for LaTeX
tools/	Custom processing scripts

3 LaTeX Primer

This section is a brief introduction to LaTeX concepts and syntax, to provide authors enough information to author documents productively without having to become "TeXnicians."

LaTeX 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.

XXX This section will discuss what the markup looks like, and explain the difference between an environment and a macro.

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 more broad. 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.

5.1 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 *feature*`desc`, and the non-indexing variants are named *feature*`descni`. 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{datadesc}{`*name*`}`
: 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*`}`
: Like `datadesc`, but without creating any index entries.

`\begin{excdesc}{`*name*`}`
: Describe an exception. This may be either a string exception or a class exception.

`\begin{funcdesc}{`*name*`}{`*parameters*`}`
: 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*`}`
Like `funcdesc`, but without creating any index entries.

`\begin{classdesc}{`*name*`}{`*constructor parameters*`}`
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*`}`
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*`}`
Like `memberdesc`, but without creating any index entries.

`\begin{methoddesc}[`*type name*`]{`*name*`}{`*parameters*`}`
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*`}`
Like `methoddesc`, but without creating any index entries.

5.2 Inline Markup

`\bfcode{`*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`.

`\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.)

`\email{`*address*`}`
An email address. Note that this is *not* hyperlinked in any of the possible output formats.

`\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.

`\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.

`\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.

`\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.

`\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.

`\strong{`*text*`}`
Strongly emphasized text; this will be presented using a bold font.

`\var{`*name*`}`
The name of a variable or formal parameter in running text.

`\version`
The version number for the documentation, as specified using `\release` in the preamble.

5.3 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}
```

`\declaremodule[`*key*`]{`*type*`}{`*name*`}`
Requires two parameters: module type (standard, builtin, extension), 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. 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 `\localmoduletable`. 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.

5.4 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.

\localmoduletable
: If a '.syn' file exists for the current chapter (or for the entire document in `howto` documents), a `synopsistable` is created with the contents loaded from the '.syn' file.

5.5 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 `table`*cols*, where *cols* is the number of columns in the table specified in lowercase Roman numerals. Within each of these environments, an additional macro, `\line`*cols*, 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.

\begin{tableii}{*colspec*}{*col1font*}{*heading1*}{*heading2*}
: 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 *col1font* parameter is used as a stylistic treatment of the first column of the table: the first column is presented as *col1font*{column1}. To avoid treating the first column specially, *col1font* may be '`textrm`'. The column headings are taken from the values *heading1* and *heading2*.

\lineii{*column1*}{*column2*}
: Create a single table row within a `tableii` environment. The text for the first column will be generated by applying the macro named by the *col1font* value when the `tableii` was opened.

\begin{tableiii}{*colspec*}{*col1font*}{*heading1*}{*heading2*}{*heading3*}
: Like the `tableii` environment, but with a third column. The heading for the third column is given by *heading3*.

\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*}{*col1font*}{*heading1*}{*heading2*}{*heading3*}{*heading4*}
: Like the `tableiii` environment, but with a fourth column. The heading for the fourth column is given by *heading4*.

\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 use by the user, but is created by the `\localmoduletable` macro.

5.6 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.

`\begin{seealso}`
: This environment creates a "See also:" heading and defines the markup used to describe individual references.

`\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).

`\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.

5.7 Index-generating Markup

Effective index generation for technical documents can be very difficult, especially for someone familliar 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 make 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 5.1, "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 `lib\jobname.idx` to be created from the `\declaremodule` 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 LaTeX 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.

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 TEX 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 (LATEX2HTML requires one of the intermediate files it creates).

latex2html Probably the longest Perl script anyone ever attempted to maintain. This converts LATEX 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 LATEX index data to a formatted index; it should be included with all LATEX 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 pdfTEX is a relatively new variant of TEX, and is used to generate the PDF version of the manuals. It is typically installed as part of most of the large TEX distributions. **pdflatex** is pdfTEX using the LATEX format.

perl Perl is required for LATEX2HTML and one of the scripts used to post-process LATEX2HTML 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

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.