Sphinx Documentation

Release 0.4.2

Georg Brandl

August 06, 2008

CONTENTS

CHAPTER

ONE

Introduction

This is the documentation for the Sphinx documentation builder. Sphinx is a tool that translates a set of reStructuredText source files into various output formats, automatically producing cross-references, indices etc.

The focus is on hand-written documentation, rather than auto-generated API docs. Though there is limited support for that kind of docs as well (which is intended to be freely mixed with hand-written content), if you need pure API docs have a look at Epydoc, which also understands reST.

1.1 Prerequisites

Sphinx needs at least **Python 2.4** to run. If you like to have source code highlighting support, you must also install the Pygments library, which you can do via setuptools' easy_install. Sphinx should work with docutils version 0.4 or some (not broken) SVN trunk snapshot.

1.2 Setting up the documentation sources

The root directory of a documentation collection is called the *source directory*. Normally, this directory also contains the Sphinx configuration file `conf.py`, but that file can also live in another directory, the *configuration directory*. New in version 0.3: Support for a different configuration directory. Sphinx comes with a script called **sphinx-quickstart** that sets up a source directory and creates a default `conf.py` from a few questions it asks you. Just run

```
$ sphinx-quickstart
```

and answer the questions.

1.3 Running a build

A build is started with the **sphinx-build** script. It is called like this:

```
$ sphinx-build -b latex sourcedir builddir
```

where *sourcedir* is the *source directory*, and *builddir* is the directory in which you want to place the built documentation (it must be an existing directory). The *-b* option selects a builder; in this example Sphinx will build LaTeX files.

The **sphinx-build** script has several more options:

-a If given, always write all output files. The default is to only write output files for new and changed source files. (This may not apply to all builders.)

-E Don't use a saved *environment* (the structure caching all cross-references), but rebuild it completely. The default is to only read and parse source files that are new or have changed since the last run.

-d *path* Since Sphinx has to read and parse all source files before it can write an output file, the parsed source files are cached as "doctree pickles". Normally, these files are put in a directory called `.doctrees` under the build directory; with this option you can select a different cache directory (the doctrees can be shared between all builders).

-c *path* Don't look for the `conf.py` in the source directory, but use the given configuration directory instead. Note that various other files and paths given by configuration values are expected to be relative to the configuration directory, so they will have to be present at this location too. New in version 0.3.

-D *setting=value* Override a configuration value set in the `conf.py` file. (The value must be a string value.)

-N Do not do colored output. (On Windows, colored output is disabled in any case.)

-q Do not output anything on standard output, only write warnings to standard error.

-P (Useful for debugging only.) Run the Python debugger, `pdb`, if an unhandled exception occurs while building.

You can also give one or more filenames on the command line after the source and build directories. Sphinx will then try to build only these output files (and their dependencies).

CHAPTER

TWO

Sphinx concepts

2.1 Document names

Since the reST source files can have different extensions (some people like `.txt`, some like `.rst` – the extension can be configured with `source_suffix`) and different OSes have different path separators, Sphinx abstracts them: all "document names" are relative to the *source directory*, the extension is stripped, and path separators are converted to slashes. All values, parameters and suchlike referring to "documents" expect such a document name.

2.2 The TOC tree

Since reST does not have facilities to interconnect several documents, or split documents into multiple output files, Sphinx uses a custom directive to add relations between the single files the documentation is made of, as well as tables of contents. The `toctree` directive is the central element.

`..` **`toctree::`**

This directive inserts a "TOC tree" at the current location, using the individual TOCs (including "sub-TOC trees") of the documents given in the directive body (whose path is relative to the document the directive occurs in). A numeric `maxdepth` option may be given to indicate the depth of the tree; by default, all levels are included. [1]

Consider this example (taken from the Python docs' library reference index):

```
.. toctree::
   :maxdepth: 2

   intro
   strings
   datatypes
   numeric
   (many more documents listed here)
```

This accomplishes two things:

- Tables of contents from all those documents are inserted, with a maximum depth of two, that means one nested heading. `toctree` directives in those documents are also taken into account.
- Sphinx knows that the relative order of the documents `intro`, `strings` and so forth, and it knows that they are children of the shown document, the library index. From this information it generates "next chapter", "previous chapter" and "parent chapter" links.

Document titles in the `toctree` will be automatically read from the title of the referenced document. If that isn't what you want, you can give the specify an explicit title and target using a similar syntax to reST hyperlinks (and Sphinx's *cross-referencing syntax*). This looks like:

```
.. toctree::

   intro
   All about strings <strings>
   datatypes
```

The second line above will link to the `strings` document, but will use the title "All about strings" instead of the title of the `strings` document.

You can use "globbing" in toctree directives, by giving the `glob` flag option. All entries are then matched against the list of available documents, and matches are inserted into the list alphabetically. Example:

```
.. toctree::
   :glob:

   intro*
   recipe/*
   *
```

This includes first all documents whose names start with `intro`, then all documents in the `recipe` folder, then all remaining documents (except the one containing the directive, of course.) [2]

In the end, all documents in the *source directory* (or subdirectories) must occur in some `toctree` directive; Sphinx will emit a warning if it finds a file that is not included, because that means that this file will not be reachable through standard navigation. Use `unused_documents` to explicitly exclude documents from building, and `exclude_dirs` to exclude whole directories.

The "master document" (selected by `master_doc`) is the "root" of the TOC tree hierarchy. It can be used as the documentation's main page, or as a "full table of contents" if you don't give a `maxdepth` option. Changed in version 0.3: Added "globbing" option.

2.3 Special names

Sphinx reserves some document names for its own use; you should not try to create documents with these names – it will cause problems.

The special document names (and pages generated for them) are:

- `genindex`, `modindex`, `search`

  These are used for the general index, the module index, and the search page, respectively.

  The general index is populated with entries from modules, all index-generating *description units*, and from `index` directives.

  The module index contains one entry per `module` directive.

  The search page contains a form that uses the generated JSON search index and JavaScript to full-text search the generated documents for search words; it should work on every major browser that supports modern JavaScript.

- every name beginning with _

  Though only few such names are currently used by Sphinx, you should not create documents or document-containing directories with such names. (Using _ as a prefix for a custom template directory is fine.)

[1] The `maxdepth` option does not apply to the LaTeX writer, where the whole table of contents will always be presented at the begin of the document, and its depth is controlled by the `tocdepth` counter, which you can reset in your `latex_preamble` config value using e.g. `\setcounter{tocdepth}{2}`.

[2] A note on available globbing syntax: you can use the standard shell constructs `*`, `?`, `[...]` and `[!...]` with the feature that these all don't match slashes. A double star `**` can be used to match any sequence of characters *including* slashes.

CHAPTER

THREE

ReStructuredText Primer

This section is a brief introduction to reStructuredText (reST) concepts and syntax, intended to provide authors with enough information to author documents productively. Since reST was designed to be a simple, unobtrusive markup language, this will not take too long.

See Also:

The authoritative reStructuredText User Documentation.

3.1 Paragraphs

The paragraph is the most basic block in a reST document. Paragraphs are simply chunks of text separated by one or more blank lines. As in Python, indentation is significant in reST, so all lines of the same paragraph must be left-aligned to the same level of indentation.

3.2 Inline markup

The standard reST inline markup is quite simple: use

- one asterisk: `*text*` for emphasis (italics),
- two asterisks: `**text**` for strong emphasis (boldface), and
- backquotes: ``` ``text`` ``` for code samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, they have to be escaped with a backslash.

Be aware of some restrictions of this markup:

- it may not be nested,
- content may not start or end with whitespace: `* text*` is wrong,
- it must be separated from surrounding text by non-word characters. Use a backslash escaped space to work around that: `thisis\ *one*\ word`.

These restrictions may be lifted in future versions of the docutils.

reST also allows for custom "interpreted text roles"', which signify that the enclosed text should be interpreted in a specific way. Sphinx uses this to provide semantic markup and cross-referencing of identifiers, as described in the appropriate section. The general syntax is `:rolename:`content``.

3.3 Lists and Quotes

List markup is natural: just place an asterisk at the start of a paragraph and indent properly. The same goes for numbered lists; they can also be autonumbered using a # sign:

```
* This is a bulleted list.
* It has two items, the second
  item uses two lines.

1. This is a numbered list.
2. It has two items too.

#. This is a numbered list.
#. It has two items too.
```

Note that Sphinx disables the use of enumerated lists introduced by alphabetic or roman numerals, such as

```
A. First item
B. Second item
```

Nested lists are possible, but be aware that they must be separated from the parent list items by blank lines:

```
* this is
* a list

  * with a nested list
  * and some subitems

* and here the parent list continues
```

Definition lists are created as follows:

```
term (up to a line of text)
   Definition of the term, which must be indented

   and can even consist of multiple paragraphs

next term
   Description.
```

Paragraphs are quoted by just indenting them more than the surrounding paragraphs.

3.4 Source Code

Literal code blocks are introduced by ending a paragraph with the special marker `::`. The literal block must be indented, to be able to include blank lines:

```
This is a normal text paragraph. The next paragraph is a code sample::

   It is not processed in any way, except
   that the indentation is removed.

   It can span multiple lines.
```

```
This is a normal text paragraph again.
```

The handling of the `::` marker is smart:

- If it occurs as a paragraph of its own, that paragraph is completely left out of the document.
- If it is preceded by whitespace, the marker is removed.
- If it is preceded by non-whitespace, the marker is replaced by a single colon.

That way, the second sentence in the above example's first paragraph would be rendered as "The next paragraph is a code sample:".

3.5 Hyperlinks

3.5.1 External links

Use `` `Link text <http://target>`_ `` for inline web links. If the link text should be the web address, you don't need special markup at all, the parser finds links and mail addresses in ordinary text.

3.5.2 Internal links

Internal linking is done via a special reST role, see the section on specific markup, *Cross-referencing arbitrary locations*.

3.6 Sections

Section headers are created by underlining (and optionally overlining) the section title with a punctuation character, at least as long as the text:

```
=================
This is a heading
=================
```

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, for the Python documentation, this convention is used which you may follow:

- `#` with overline, for parts
- `*` with overline, for chapters
- `=`, for sections
- `-`, for subsections
- `^`, for subsubsections
- `"`, for paragraphs

Of course, you are free to use your own marker characters (see the reST documentation), and use a deeper nesting level, but keep in mind that most target formats (HTML, LaTeX) have a limited supported nesting depth.

3.7 Explicit Markup

"Explicit markup" is used in reST for most constructs that need special handling, such as footnotes, specially-highlighted paragraphs, comments, and generic directives.

An explicit markup block begins with a line starting with `..` followed by whitespace and is terminated by the next paragraph at the same level of indentation. (There needs to be a blank line between explicit markup and normal paragraphs. This may all sound a bit complicated, but it is intuitive enough when you write it.)

3.8 Directives

A directive is a generic block of explicit markup. Besides roles, it is one of the extension mechanisms of reST, and Sphinx makes heavy use of it.

Basically, a directive consists of a name, arguments, options and content. (Keep this terminology in mind, it is used in the next chapter describing custom directives.) Looking at this example,

```
.. function:: foo(x)
              foo(y, z)
   :bar: no

   Return a line of text input from the user.
```

`function` is the directive name. It is given two arguments here, the remainder of the first line and the second line, as well as one option `bar` (as you can see, options are given in the lines immediately following the arguments and indicated by the colons).

The directive content follows after a blank line and is indented relative to the directive start.

3.9 Images

reST supports an image directive, used like so:

```
.. image:: gnu.png
   (options)
```

When used within Sphinx, the file name given (here `gnu.png`) must be relative to the source file, and Sphinx will automatically copy image files over to a subdirectory of the output directory on building (e.g. the `_static` directory for HTML output.)

Sphinx extends the standard docutils behavior by allowing an asterisk for the extension:

```
.. image:: gnu.*
```

Sphinx then searches for all images matching the provided pattern and determines their type. Each builder then chooses the best image out of these candidates. For instance, if the file name `gnu.*` was given and two files `gnu.pdf` and `gnu.png` existed in the source tree, the LaTeX builder would choose the former, while the HTML builder would prefer the latter. Changed in version 0.4: Added the support for file names ending in an asterisk.

3.10 Footnotes

For footnotes, use `[#]_` to mark the footnote location, and add the footnote body at the bottom of the document after a "Footnotes" rubric heading, like so:

```
Lorem ipsum [#]_ dolor sit amet ... [#]_

.. rubric:: Footnotes

.. [#] Text of the first footnote.
.. [#] Text of the second footnote.
```

You can also explicitly number the footnotes for better context.

3.11 Comments

Every explicit markup block which isn't a valid markup construct (like the footnotes above) is regarded as a comment.

3.12 Source encoding

Since the easiest way to include special characters like em dashes or copyright signs in reST is to directly write them as Unicode characters, one has to specify an encoding:

All documentation source files must be in UTF-8 encoding, and the HTML documents written from them will be in that encoding as well.

3.13 Gotchas

There are some problems one commonly runs into while authoring reST documents:

- **Separation of inline markup:** As said above, inline markup spans must be separated from the surrounding text by non-word characters, you have to use an escaped space to get around that.

CHAPTER

FOUR

Sphinx Markup Constructs

Sphinx adds a lot of new directives and interpreted text roles to standard reST markup. This section contains the reference material for these facilities.

4.1 Module-specific markup

The markup described in this section is used to provide information about a module being documented. Normally this markup appears after a title heading; a typical module section might start like this:

```
:mod:`parrot` -- Dead parrot access
===================================

.. module:: parrot
   :platform: Unix, Windows
   :synopsis: Analyze and reanimate dead parrots.
.. moduleauthor:: Eric Cleese <eric@python.invalid>
.. moduleauthor:: John Idle <john@python.invalid>
```

The directives you can use for module are:

.. **module:: name**
This directive marks the beginning of the description of a module (or package submodule, in which case the name should be fully qualified, including the package name). It does not create content (like e.g. `class` does).

This directive will also cause an entry in the global module index.

The `platform` option, if present, is a comma-separated list of the platforms on which the module is available (if it is available on all platforms, the option should be omitted). 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.

The `synopsis` option should consist of one sentence describing the module's purpose – it is currently only used in the Global Module Index.

The `deprecated` option can be given (with no value) to mark a module as deprecated; it will be designated as such in various locations then.

.. **currentmodule:: name**
This directive tells Sphinx that the classes, functions etc. documented from here are in the given module (like `module`), but it will not create index entries, an entry in the Global Module Index, or a link target for `mod`. This is helpful in situations where documentation for things in a module is spread over multiple files or sections – one location has the `module` directive, the others only `currentmodule`.

.. **moduleauthor:: name <email>**
The `moduleauthor` directive, which can appear multiple times, names the authors of the module code, just

like `sectionauthor` names the author(s) of a piece of documentation. It too only produces output if the `show_authors` configuration value is True.

Note: It is important to make the section title of a module-describing file meaningful since that value will be inserted in the table-of-contents trees in overview files.

4.2 Description units

There are a number of directives used to describe specific features provided by modules. Each directive requires one or more signatures to provide basic information about what is being described, and the content should be the description. The basic version makes entries in the general index; if no index entry is desired, you can give the directive option flag `:noindex:`. The following example shows all of the features of this directive type:

```
.. function:: spam(eggs)
              ham(eggs)
   :noindex:

   Spam or ham the foo.
```

The signatures of object methods or data attributes should always include the type name (`.. method:: FileInput.input(...)`), even if it is obvious from the context which type they belong to; this is to enable consistent cross-references. If you describe methods belonging to an abstract protocol, such as "context managers", include a (pseudo-)type name too to make the index entries more informative.

The directives are:

`..` **`cfunction:: type name(signature)`**
Describes a C function. The signature should be given as in C, e.g.:

```
.. cfunction:: PyObject* PyType_GenericAlloc(PyTypeObject *type, Py_ssize_t nitems)
```

This is also used to describe function-like preprocessor macros. The names of the arguments should be given so they may be used in the description.

Note that you don't have to backslash-escape asterisks in the signature, as it is not parsed by the reST inliner.

`..` **`cmember:: type name`**
Describes a C struct member. Example signature:

```
.. cmember:: PyObject* PyTypeObject.tp_bases
```

The text of the description should include the range of values allowed, how the value should be interpreted, and whether the value can be changed. References to structure members in text should use the `member` role.

`..` **`cmacro:: name`**
Describes a "simple" C macro. Simple macros are macros which are used for code expansion, but which do not take arguments so cannot be described as functions. This is not to be used for simple constant definitions. Examples of its use in the Python documentation include `PyObject_HEAD` and `Py_BEGIN_ALLOW_THREADS`.

`..` **`ctype:: name`**
Describes a C type. The signature should just be the type name.

`..` **`cvar:: type name`**
Describes a global C variable. The signature should include the type, such as:

```
.. cvar:: PyObject* PyClass_Type
```

.. **data:: name**
Describes global data in a module, including both variables and values used as "defined constants." Class and object attributes are not documented using this environment.

.. **exception:: name**
Describes an exception class. The signature can, but need not include parentheses with constructor arguments.

.. **function:: name(signature)**
Describes a module-level function. The signature should include the parameters, enclosing optional parameters in brackets. Default values can be given if it enhances clarity; see *Signatures*. For example:

```
.. function:: Timer.repeat([repeat=3[, number=1000000]])
```

Object methods are not documented using this directive. 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.

.. **class:: name[(signature)]**
Describes a class. The signature can include parentheses with parameters which will be shown as the constructor arguments. See also *Signatures*.

Methods and attributes belonging to the class should be placed in this directive's body. If they are placed outside, the supplied name should contain the class name so that cross-references still work. Example:

```
.. class:: Foo
   .. method:: quux()

-- or --

.. class:: Bar

.. method:: Bar.quux()
```

The first way is the preferred one. New in version 0.4: The standard reST directive `class` is now provided by Sphinx under the name `cssclass`.

.. **attribute:: name**
Describes 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.

.. **method:: name(signature)**
Describes an object method. The parameters should not include the `self` parameter. The description should include similar information to that described for `function`. See also *Signatures*.

.. **staticmethod:: name(signature)**
Like `method`, but indicates that the method is a static method. New in version 0.4.

.. **opcode:: name**
Describes a Python bytecode instruction (this is not very useful for projects other than Python itself).

.. **cmdoption:: name args, name args, ...**
Describes a command line option or switch. Option argument names should be enclosed in angle brackets. Example:

```
.. cmdoption:: -m <module>, --module <module>

   Run a module as a script.
```

The directive will create a cross-reference target named after the *first* option, referencable by `option` (in the example case, you'd use something like `:option:'-m'`).

`..` **`envvar:: name`**
Describes an environment variable that the documented code uses or defines.

There is also a generic version of these directives:

`..` **`describe:: text`**
This directive produces the same formatting as the specific ones explained above but does not create index entries or cross-referencing targets. It is used, for example, to describe the directives in this document. Example:

```
.. describe:: opcode

   Describes a Python bytecode instruction.
```

Extensions may add more directives like that, using the `add_description_unit()` method.

4.2.1 Signatures

Signatures of functions, methods and class constructors can be given like they would be written in Python, with the exception that optional parameters can be indicated by brackets:

```
.. function:: compile(source[, filename[, symbol]])
```

It is customary to put the opening bracket before the comma. In addition to this "nested" bracket style, a "flat" style can also be used, due to the fact that most optional parameters can be given independently:

```
.. function:: compile(source[, filename, symbol])
```

Default values for optional arguments can be given (but if they contain commas, they will confuse the signature parser). Python 3-style argument annotations can also be given as well as return type annotations:

```
.. function:: compile(source : string[, filename, symbol]) -> ast object
```

4.2.2 Info field lists

New in version 0.4. Inside description unit directives, reST field lists with these fields are recognized and formatted nicely:

- `param`, `parameter`, `arg`, `argument`, `key`, `keyword`: Description of a parameter.
- `type`: Type of a parameter.
- `raises`, `raise`, `except`, `exception`: That (and when) a specific exception is raised.
- `var`, `ivar`, `cvar`: Description of a variable.
- `returns`, `return`: Description of the return value.
- `rtype`: Return type.

The field names must consist of one of these keywords and an argument (except for `returns` and `rtype`, which do not need an argument). This is best explained by an example:

```
.. function:: format_exception(etype, value, tb[, limit=None])

   Format the exception with a traceback.

   :param object: exception type
   :param value: exception value
   :param tb: traceback object
   :param limit: maximum number of stack frames to show
   :type limit: integer or None
   :rtype: list of strings
```

This will render like this:

format_exception(*etype, value, tb, [limit=None]*)
Format the exception with a traceback.

Parameters
- *object* – exception type
- *value* – exception value
- *tb* – traceback object
- *limit* (integer or None) – maximum number of stack frames to show

Return type list of strings

4.3 Paragraph-level markup

These directives create short paragraphs and can be used inside information units as well as normal text:

.. note::
An especially important bit of information about an API that a user should be aware of when using whatever bit of API the note pertains to. The content of the directive should be written in complete sentences and include all appropriate punctuation.

Example:

```
.. note::

   This function is not suitable for sending spam e-mails.
```

.. warning::
An important bit of information about an API that a user should be very aware of when using whatever bit of API the warning pertains to. The content of the directive should be written in complete sentences and include all appropriate punctuation. This differs from `note` in that it is recommended over `note` for information regarding security.

.. versionadded:: version
This directive documents the version of the project which added the described feature to the library or C API. When this applies to an entire module, it should be placed at the top of the module section before any prose.

The first argument must be given and is the version in question; you can add a second argument consisting of a *brief* explanation of the change.

Example:

```
.. versionadded:: 2.5
   The 'spam' parameter.
```

Note that there must be no blank line between the directive head and the explanation; this is to make these blocks visually continuous in the markup.

.. **versionchanged:: version**
Similar to `versionadded`, but describes when and what changed in the named feature in some way (new parameters, changed side effects, etc.).

.. **seealso::**
Many sections include a list of references to module documentation or external documents. These lists are created using the `seealso` directive.

The `seealso` directive is typically placed in a section just before any sub-sections. For the HTML output, it is shown boxed off from the main flow of the text.

The content of the `seealso` directive should be a reST definition list. Example:

```
.. seealso::

   Module :mod:`zipfile`
      Documentation of the :mod:`zipfile` standard module.

   `GNU tar manual, Basic Tar Format <http://link>`_
      Documentation for tar archive files, including GNU tar extensions.
```

.. **rubric:: title**
This directive creates a paragraph heading that is not used to create a table of contents node.

Note: If the *title* of the rubric is "Footnotes", this rubric is ignored by the LaTeX writer, since it is assumed to only contain footnote definitions and therefore would create an empty heading.

.. **centered::**
This directive creates a centered boldfaced paragraph. Use it as follows:

```
.. centered::

   Paragraph contents.
```

4.4 Table-of-contents markup

The `toctree` directive, which generates tables of contents of subdocuments, is described in "Sphinx concepts".

For local tables of contents, use the standard reST `contents` directive.

4.5 Index-generating markup

Sphinx automatically creates index entries from all information units (like functions, classes or attributes) like discussed before.

However, there is also an explicit directive available, to make the index more comprehensive and enable index entries in documents where information is not mainly contained in information units, such as the language reference.

.. **index:: <entries>**
This directive contains one or more index entries. Each entry consists of a type and a value, separated by a colon.

For example:

```
.. index::
   single: execution; context
   module: __main__
   module: sys
   triple: module; search; path

The execution context
---------------------

...
```

This directive contains five entries, which will be converted to entries in the generated index which link to the exact location of the index statement (or, in case of offline media, the corresponding page number).

Since index directives generate cross-reference targets at their location in the source, it makes sense to put them *before* the thing they refer to – e.g. a heading, as in the example above.

The possible entry types are:

single Creates a single index entry. Can be made a subentry by separating the subentry text with a semicolon (this notation is also used below to describe what entries are created).

pair `pair: loop; statement` is a shortcut that creates two index entries, namely `loop; statement` and `statement; loop`.

triple Likewise, `triple: module; search; path` is a shortcut that creates three index entries, which are `module; search path`, `search; path, module` and `path; module search`.

module, keyword, operator, object, exception, statement, builtin These all create two index entries. For example, `module: hashlib` creates the entries `module; hashlib` and `hashlib; module`.

For index directives containing only "single" entries, there is a shorthand notation:

```
.. index:: BNF, grammar, syntax, notation
```

This creates four index entries.

4.6 Glossary

`.. glossary::`
This directive must contain a reST definition list with terms and definitions. The definitions will then be referencable with the `term` role. Example:

```
.. glossary::

   environment
      A structure where information about all documents under the root is saved,
      and used for cross-referencing.  The environment is pickled after the
      parsing stage, so that successive runs only need to read and parse new and
      changed documents.

   source directory
      The directory which, including its subdirectories, contains all source
      files for one Sphinx project.
```

4.7 Grammar production displays

Special markup is available for displaying the productions of a formal grammar. The markup is simple and does not attempt to model all aspects of BNF (or any derived forms), but provides enough to allow context-free grammars to be displayed in a way that causes uses of a symbol to be rendered as hyperlinks to the definition of the symbol. There is this directive:

.. **productionlist::**
   This directive is used to enclose a group of productions. Each production is given on a single line and consists of a name, separated by a colon from the following definition. If the definition spans multiple lines, each continuation line must begin with a colon placed at the same column as in the first line.

   Blank lines are not allowed within `productionlist` directive arguments.

   The definition can contain token names which are marked as interpreted text (e.g. `sum ::= 'integer' "+" 'integer'`) – this generates cross-references to the productions of these tokens.

   Note that no further reST parsing is done in the production, so that you don't have to escape `*` or `|` characters.

The following is an example taken from the Python Reference Manual:

```
.. productionlist::
   try_stmt: try1_stmt | try2_stmt
   try1_stmt: "try" ":" 'suite'
            : ("except" ['expression' ["," 'target']] ":" 'suite')+
            : ["else" ":" 'suite']
            : ["finally" ":" 'suite']
   try2_stmt: "try" ":" 'suite'
            : "finally" ":" 'suite'
```

4.8 Showing code examples

Examples of Python source code or interactive sessions are represented using standard reST literal blocks. They are started by a `::` at the end of the preceding paragraph and delimited by indentation.

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
>>>
```

Syntax highlighting is done with Pygments (if it's installed) and handled in a smart way:

- There is a "highlighting language" for each source file. Per default, this is `'python'` as the majority of files will have to highlight Python snippets.
- Within Python highlighting mode, interactive sessions are recognized automatically and highlighted appropriately.
- The highlighting language can be changed using the `highlight` directive, used as follows:

  ```
  .. highlight:: c
  ```

  This language is used until the next `highlight` directive is encountered.

- For documents that have to show snippets in different languages, there's also a `code-block` directive that is given the highlighting language directly:

  ```
  .. code-block:: ruby

     Some Ruby code.
  ```

  The directive's alias name `sourcecode` works as well.

- The valid values for the highlighting language are:
  - `none` (no highlighting)
  - `python` (the default)
  - `rest`
  - `c`
  - ... and any other lexer name that Pygments supports.

- If highlighting with the selected language fails, the block is not highlighted in any way.

4.8.1 Line numbers

If installed, Pygments can generate line numbers for code blocks. For automatically-highlighted blocks (those started by `::`), line numbers must be switched on in a `highlight` directive, with the `linenothreshold` option:

```
.. highlight:: python
   :linenothreshold: 5
```

This will produce line numbers for all code blocks longer than five lines.

For `code-block` blocks, a `linenos` flag option can be given to switch on line numbers for the individual block:

```
.. code-block:: ruby
   :linenos:

   Some more Ruby code.
```

4.8.2 Includes

`.. literalinclude:: filename`

Longer displays of verbatim text may be included by storing the example text in an external file containing only plain text. The file may be included using the `literalinclude` directive. [1] For example, to include the Python source file `example.py`, use:

```
.. literalinclude:: example.py
```

The file name is relative to the current file's path.

The directive also supports the `linenos` flag option to switch on line numbers, and a `language` option to select a language different from the current file's standard language. Example with options:

```
.. literalinclude:: example.rb
   :language: ruby
   :linenos:
```

[1] There is a standard `.. include` directive, but it raises errors if the file is not found. This one only emits a warning.

4.9 Inline markup

Sphinx uses interpreted text roles to insert semantic markup into documents.

Variable names are an exception, they should be marked simply with `*var*`.

For all other roles, you have to write `:rolename:`content``.

Note: The default role (`content`) has no special meaning by default. You are free to use it for anything you like.

4.9.1 Cross-referencing syntax

Cross-references are generated by many semantic interpreted text roles. Basically, you only need to write `:role:`target``, and a link will be created to the item named *target* of the type indicated by *role*. The links's text will be the same as *target*.

There are some additional facilities, however, that make cross-referencing roles more versatile:

- You may supply an explicit title and reference target, like in reST direct hyperlinks: `:role:`title <target>`` will refer to *target*, but the link text will be *title*.
- If you prefix the content with `!`, no reference/hyperlink will be created.
- For the Python object roles, if you prefix the content with `~`, the link text will only be the last component of the target. For example, `:meth:`~Queue.Queue.get`` will refer to `Queue.Queue.get` but only display `get` as the link text.

  In HTML output, the link's `title` attribute (that is e.g. shown as a tool-tip on mouse-hover) will always be the full target name.

4.9.2 Cross-referencing Python objects

The following roles refer to objects in modules and are possibly hyperlinked if a matching identifier is found:

:mod:
The name of a module; a dotted name may be used. This should also be used for package names.

:func:
The name of a Python function; dotted names may be used. The role text needs not include trailing parentheses to enhance readability; they will be added automatically by Sphinx if the `add_function_parentheses` config value is true (the default).

:data:
The name of a module-level variable.

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

:class:
A class name; a dotted name may be used.

:meth:
The name of a method of an object. The role text should include the type name and the method name; if it occurs within the description of a type, the type name can be omitted. A dotted name may be used.

:attr:
The name of a data attribute of an object.

:exc:
The name of an exception. A dotted name may be used.

:obj:
The name of an object of unspecified type. Useful e.g. as the `default_role`. New in version 0.4.

The name enclosed in this markup can include a module name and/or a class name. For example, `:func:`filter`` could refer to a function named `filter` in the current module, or the built-in function of that name. In contrast, `:func:`foo.filter`` clearly refers to the `filter` function in the `foo` module.

Normally, names in these roles are searched first without any further qualification, then with the current module name prepended, then with the current module and class name (if any) prepended. If you prefix the name with a dot, this order is reversed. For example, in the documentation of Python's `codecs` module, `:func:`open`` always refers to the built-in function, while `:func:`.open`` refers to `codecs.open()`.

A similar heuristic is used to determine whether the name is an attribute of the currently documented class.

4.9.3 Cross-referencing C constructs

The following roles create cross-references to C-language constructs if they are defined in the documentation:

:cdata:
The name of a C-language variable.

:cfunc:
The name of a C-language function. Should include trailing parentheses.

:cmacro:
The name of a "simple" C macro, as defined above.

:ctype:
The name of a C-language type.

4.9.4 Cross-referencing other items of interest

The following roles do possibly create a cross-reference, but do not refer to objects:

:envvar:
An environment variable. Index entries are generated. Also generates a link to the matching `envvar` directive, if it exists.

:token:
The name of a grammar token (used to create links between `productionlist` directives).

:keyword:
The name of a keyword in Python. This creates a link to a reference label with that name, if it exists.

:option:
A command-line option to an executable program. The leading hyphen(s) must be included. This generates a link to a `cmdoption` directive, if it exists.

The following role creates a cross-reference to the term in the glossary:

:term:
Reference to a term in the glossary. The glossary is created using the `glossary` directive containing a definition list with terms and definitions. It does not have to be in the same file as the `term` markup, for example the Python docs have one global glossary in the `glossary.rst` file.

If you use a term that's not explained in a glossary, you'll get a warning during build.

4.9.5 Cross-referencing arbitrary locations

To support cross-referencing to arbitrary locations in the documentation, the standard reST labels used. Of course, for this to work label names must be unique throughout the entire documentation. There are two ways in which you can refer to labels:

- If you place a label directly before a section title, you can reference to it with `:ref:`label-name``. Example:

  ```
  .. _my-reference-label:

  Section to cross-reference
  --------------------------

  This is the text of the section.

  It refers to the section itself, see :ref:`my-reference-label`.
  ```

  The `:ref:` role would then generate a link to the section, with the link title being "Section to cross-reference".

- Labels that aren't placed before a section title can still be referenced to, but you must give the link an explicit title, using this syntax: `:ref:`Link title <label-name>``.

4.9.6 Other semantic markup

The following roles don't do anything special except formatting the text in a different style:

`:command:`
The name of an OS-level command, such as `rm`.

`:dfn:`
Mark the defining instance of a term in the text. (No index entries are generated.)

`:file:`
The name of a file or directory. Within the contents, you can use curly braces to indicate a "variable" part, for example:

```
... is installed in :file:`/usr/lib/python2.{x}/site-packages` ...
```

In the built documentation, the `x` will be displayed differently to indicate that it is to be replaced by the Python minor version.

`:guilabel:`
Labels presented as part of an interactive user interface should be marked using `guilabel`. This includes labels from text-based interfaces such as those created using `curses` or other text-based libraries. Any label used in the interface should be marked with this role, including button labels, window titles, field names, menu and menu selection names, and even values in selection lists.

`:kbd:`
Mark a sequence of keystrokes. What form the key sequence takes may depend on platform- or application-specific conventions. When there are no relevant conventions, the names of modifier keys should be spelled out, to improve accessibility for new users and non-native speakers. For example, an *xemacs* key sequence may be marked like `:kbd:`C-x C-f``, but without reference to a specific application or platform, the same sequence should be marked as `:kbd:`Control-x Control-f``.

`:mailheader:`
The name of an RFC 822-style mail header. This markup does not imply that the header is being used in an email message, but can be used to refer to any header of the same "style." This is also used for headers defined

by the various MIME specifications. The header name should be entered in the same way it would normally be found in practice, with the camel-casing conventions being preferred where there is more than one common usage. For example: `:mailheader:'Content-Type'`.

:makevar:
The name of a **make** variable.

:manpage:
A reference to a Unix manual page including the section, e.g. `:manpage:'ls(1)'`.

:menuselection:
Menu selections should be marked using the `menuselection` role. This is used to mark a complete sequence of menu selections, including selecting submenus and choosing a specific operation, or any subsequence of such a sequence. The names of individual selections should be separated by `-->`.

For example, to mark the selection "Start > Programs", use this markup:

```
:menuselection:'Start --> Programs'
```

When including a selection that includes some trailing indicator, such as the ellipsis some operating systems use to indicate that the command opens a dialog, the indicator should be omitted from the selection name.

:mimetype:
The name of a MIME type, or a component of a MIME type (the major or minor portion, taken alone).

:newsgroup:
The name of a Usenet newsgroup.

:program:
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 Windows programs.

:regexp:
A regular expression. Quotes should not be included.

:samp:
A piece of literal text, such as code. Within the contents, you can use curly braces to indicate a "variable" part, as in `:file:`.

If you don't need the "variable part" indication, use the standard `"code"` instead.

The following roles generate external links:

:pep:
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.

:rfc:
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.

Note that there are no special roles for including hyperlinks as you can use the standard reST markup for that purpose.

4.9.7 Substitutions

The documentation system provides three substitutions that are defined by default. They are set in the build configuration file.

|release|
Replaced by the project release the documentation refers to. This is meant to be the full version string including alpha/beta/release candidate tags, e.g. `2.5.2b3`. Set by `release`.

|version|
Replaced by the project version the documentation refers to. This is meant to consist only of the major and minor version parts, e.g. `2.5`, even for version 2.5.1. Set by `version`.

|today|
Replaced by either today's date, or the date set in the build configuration file. Normally has the format `April 14, 2007`. Set by `today_fmt` and `today`.

4.10 Miscellaneous markup

4.10.1 File-wide metadata

reST has the concept of "field lists"; these are a sequence of fields marked up like this:

```
:Field name: Field content
```

A field list at the very top of a file is parsed as the "docinfo", which in normal documents can be used to record the author, date of publication and other metadata. In Sphinx, the docinfo is used as metadata, too, but not displayed in the output.

At the moment, these metadata fields are recognized:

tocdepth The maximum depth for a table of contents of this file. New in version 0.4.

nocomments If set, the web application won't display a comment form for a page generated from this source file.

4.10.2 Meta-information markup

.. sectionauthor::
Identifies the author of the current section. The argument should include the author's name such that it can be used for presentation and email address. The domain name portion of the address should be lower case. Example:

```
.. sectionauthor:: Guido van Rossum <guido@python.org>
```

By default, this markup isn't reflected in the output in any way (it helps keep track of contributions), but you can set the configuration value `show_authors` to True to make them produce a paragraph in the output.

4.10.3 Tables

Use standard reStructuredText tables. They work fine in HTML output, however there are some gotchas when using tables in LaTeX: the column width is hard to determine correctly automatically. For this reason, the following directive exists:

.. tabularcolumns:: column spec
This directive gives a "column spec" for the next table occurring in the source file. The spec is the second argument to the LaTeX `tabulary` package's environment (which Sphinx uses to translate tables). It can have values like

```
|l|l|l|
```

which means three left-adjusted, nonbreaking columns. For columns with longer text that should automatically be broken, use either the standard `p{width}` construct, or tabulary's automatic specifiers:

`L`	ragged-left column with automatic width
`R`	ragged-right column with automatic width
`C`	centered column with automatic width
`J`	justified column with automatic width

The automatic width is determined by rendering the content in the table, and scaling them according to their share of the total width.

By default, Sphinx uses a table layout with `L` for every column. New in version 0.3.

Warning: Tables that contain literal blocks cannot be set with `tabulary`. They are therefore set with the standard LaTeX `tabular` environment. Also, the verbatim environment used for literal blocks only works in `p{width}` columns, which means that by default, Sphinx generates such column specs for such tables. Use the `tabularcolumns` directive to get finer control over such tables.

CHAPTER

FIVE

Available builders

These are the built-in Sphinx builders. More builders can be added by *extensions*.

The builder's "name" must be given to the **-b** command-line option of **sphinx-build** to select a builder.

class **StandaloneHTMLBuilder**()
This is the standard HTML builder. Its output is a directory with HTML files, complete with style sheets and optionally the reST sources. There are quite a few configuration values that customize the output of this builder, see the chapter *Options for HTML output* for details.

Its name is `html`.

class **HTMLHelpBuilder**()
This builder produces the same output as the standalone HTML builder, but also generates HTML Help support files that allow the Microsoft HTML Help Workshop to compile them into a CHM file.

Its name is `htmlhelp`.

class **PickleHTMLBuilder**()
This builder produces a directory with pickle files containing mostly HTML fragments and TOC information, for use of a web application (or custom postprocessing tool) that doesn't use the standard HTML templates. It also is the format used by the Sphinx Web application.

See *Pickle builder details* for details about the output format.

Its name is `pickle`. (The old name `web` still works as well.)

class **LaTeXBuilder**()
This builder produces a bunch of LaTeX files in the output directory. You have to specify which documents are to be included in which LaTeX files via the `latex_documents` configuration value. There are a few configuration values that customize the output of this builder, see the chapter *Options for LaTeX output* for details.

Its name is `latex`.

class **TextBuilder**()
This builder produces a text file for each reST file – this is almost the same as the reST source, but with much of the markup stripped for better readability.

Its name is `text`. New in version 0.4.

class **ChangesBuilder**()
This builder produces an HTML overview of all `versionadded`, `versionchanged` and `deprecated` directives for the current `version`. This is useful to generate a ChangeLog file, for example.

Its name is `changes`.

class **CheckExternalLinksBuilder**()
This builder scans all documents for external links, tries to open them with `urllib2`, and writes an overview which ones are broken and redirected to standard output and to `output.txt` in the output directory.

Its name is `linkcheck`.

Built-in Sphinx extensions that offer more builders are:

- `doctest`
- `coverage`

5.1 Pickle builder details

The builder outputs one pickle file per source file, and a few special files. It also copies the reST source files in the directory `_sources` under the output directory.

The files per source file have the extensions `.fpickle`, and are arranged in directories just as the source files are. They unpickle to a dictionary with these keys:

`body` The HTML "body" (that is, the HTML rendering of the source file), as rendered by the HTML translator.

`title` The title of the document, as HTML (may contain markup).

`toc` The table of contents for the file, rendered as an HTML `<ul>`.

`display_toc` A boolean that is `True` if the `toc` contains more than one entry.

`current_page_name` The document name of the current file.

`parents`, `prev` and `next` Information about related chapters in the TOC tree. Each relation is a dictionary with the keys `link` (HREF for the relation) and `title` (title of the related document, as HTML). `parents` is a list of relations, while `prev` and `next` are a single relation.

`sourcename` The name of the source file under `_sources`.

The special files are located in the root output directory. They are:

`environment.pickle` The build environment. (XXX add important environment properties)

`globalcontext.pickle` A pickled dict with these keys:

`project`, `copyright`, `release`, `version` The same values as given in the configuration file.

`style`, `use_modindex` `html_style` and `html_use_modindex`, respectively.

`last_updated` Date of last build.

`builder` Name of the used builder, in the case of pickles this is always `'pickle'`.

`titles` A dictionary of all documents' titles, as HTML strings.

`searchindex.pickle` An index that can be used for searching the documentation. It is a pickled list with these entries:

- A list of indexed docnames.
- A list of document titles, as HTML strings, in the same order as the first list.
- A dict mapping word roots (processed by an English-language stemmer) to a list of integers, which are indices into the first list.

CHAPTER

SIX

The build configuration file

The *configuration directory* must contain a file named `conf.py`. This file (containing Python code) is called the "build configuration file" and contains all configuration needed to customize Sphinx input and output behavior.

The configuration file if executed as Python code at build time (using `execfile()`, and with the current directory set to its containing directory), and therefore can execute arbitrarily complex code. Sphinx then reads simple names from the file's namespace as its configuration.

Important points to note:

- If not otherwise documented, values must be strings, and their default is the empty string.
- The term "fully-qualified name" refers to a string that names an importable Python object inside a module; for example, the FQN `"sphinx.builder.Builder"` means the `Builder` class in the `sphinx.builder` module.
- Remember that document names use `/` as the path separator and don't contain the file name extension.
- Since `conf.py` is read as a Python file, the usual rules apply for encodings and Unicode support: declare the encoding using an encoding cookie (a comment like `# -*- coding: utf-8 -*-`) and use Unicode string literals when you include non-ASCII characters in configuration values.
- The contents of the config namespace are pickled (so that Sphinx can find out when configuration changes), so it may not contain unpickleable values – delete them from the namespace with `del` if appropriate. Modules are removed automatically, so you don't need to `del` your imports after use.

6.1 General configuration

`extensions`

A list of strings that are module names of Sphinx extensions. These can be extensions coming with Sphinx (named `sphinx.addons.*`) or custom ones.

Note that you can extend `sys.path` within the conf file if your extensions live in another directory – but make sure you use absolute paths. If your extension path is relative to the *configuration directory*, use `os.path.abspath()` like so:

```
import sys, os

sys.path.append(os.path.abspath('sphinxext'))

extensions = ['extname']
```

That way, you can load an extension called `extname` from the subdirectory `sphinxext`.

The configuration file itself can be an extension; for that, you only need to provide a `setup()` function in it.

source_suffix
The file name extension of source files. Only files with this suffix will be read as sources. Default is `.rst`.

master_doc
The document name of the "master" document, that is, the document that contains the root `toctree` directive. Default is `'contents'`.

project
The documented project's name.

copyright
A copyright statement in the style `'2008, Author Name'`.

version
The major project version, used as the replacement for `|version|`. For example, for the Python documentation, this may be something like `2.6`.

release
The full project version, used as the replacement for `|release|` and e.g. in the HTML templates. For example, for the Python documentation, this may be something like `2.6.0rc1`.

If you don't need the separation provided between `version` and `release`, just set them both to the same value.

today
today_fmt
These values determine how to format the current date, used as the replacement for `|today|`.

- If you set `today` to a non-empty value, it is used.
- Otherwise, the current time is formatted using `time.strftime()` and the format given in `today_fmt`.

The default is no `today` and a `today_fmt` of `'%B %d, %Y'`.

unused_docs
A list of document names that are present, but not currently included in the toctree. Use this setting to suppress the warning that is normally emitted in that case.

exclude_dirs
A list of directory names, relative to the source directory, that are to be excluded from the search for source files. New in version 0.3.

exclude_trees
A list of directory names, relative to the source directory, that are to be recursively exlucded from the search for source files, that is, their subdirectories won't be searched too. New in version 0.4.

pygments_style
The style name to use for Pygments highlighting of source code. Default is `'sphinx'`, which is a builtin style designed to match Sphinx' default style. Changed in version 0.3: If the value is a fully-qualified name of a custom Pygments style class, this is then used as custom style.

templates_path
A list of paths that contain extra templates (or templates that overwrite builtin templates). Relative paths are taken as relative to the configuration directory.

template_bridge
A string with the fully-qualified name of a callable (or simply a class) that returns an instance of `TemplateBridge`. This instance is then used to render HTML documents, and possibly the output of other builders (currently the changes builder).

default_role
The name of a reST role (builtin or Sphinx extension) to use as the default role, that is, for text marked up

`'like this'`. This can be set to `'obj'` to make `'filter'` a cross-reference to the function "filter". The default is `None`, which doesn't reassign the default role.

The default role can always be set within individual documents using the standard reST `default-role` directive. New in version 0.4.

add_function_parentheses
A boolean that decides whether parentheses are appended to function and method role text (e.g. the content of `:func:'input'`) to signify that the name is callable. Default is `True`.

add_module_names
A boolean that decides whether module names are prepended to all *description unit* titles, e.g. for `function` directives. Default is `True`.

show_authors
A boolean that decides whether `moduleauthor` and `sectionauthor` directives produce any output in the built files.

6.2 Options for HTML output

These options influence HTML as well as HTML Help output, and other builders that use Sphinx' HTMLWriter class.

html_title
The "title" for HTML documentation generated with Sphinx' own templates. This is appended to the `<title>` tag of individual pages, and used in the navigation bar as the "topmost" element. It defaults to `'<project> v<revision> documentation'`, where the placeholders are replaced by the config values of the same name.

html_short_title
A shorter "title" for the HTML docs. This is used in for links in the header and in the HTML Help docs. If not given, it defaults to the value of `html_title`. New in version 0.4.

html_style
The style sheet to use for HTML pages. A file of that name must exist either in Sphinx' `static/` path, or in one of the custom paths given in `html_static_path`. Default is `'default.css'`.

html_logo
If given, this must be the name of an image file that is the logo of the docs. It is placed at the top of the sidebar; its width should therefore not exceed 200 pixels. Default: `None`. New in version 0.4.1: The image file will be copied to the `_static` directory of the output HTML, so an already existing file with that name will be overwritten.

html_favicon
If given, this must be the name of an image file (within the static path, see below) that is the favicon of the docs. Modern browsers use this as icon for tabs, windows and bookmarks. It should be a Windows-style icon file (`.ico`), which is 16x16 or 32x32 pixels large. Default: `None`. New in version 0.4.

html_static_path
A list of paths that contain custom static files (such as style sheets or script files). Relative paths are taken as relative to the configuration directory. They are copied to the output directory after the builtin static files, so a file named `default.css` will overwrite the builtin `default.css`. Changed in version 0.4: The paths in `html_static_path` can now contain subdirectories.

html_last_updated_fmt
If this is not the empty string, a 'Last updated on:' timestamp is inserted at every page bottom, using the given `strftime()` format. Default is `'%b %d, %Y'`.

html_use_smartypants
If true, *SmartyPants* will be used to convert quotes and dashes to typographically correct entities. Default:

True.

html_sidebars
Custom sidebar templates, must be a dictionary that maps document names to template names. Example:

```
html_sidebars = {
   'using/windows': 'windowssidebar.html'
}
```

This will render the template windowssidebar.html within the sidebar of the given document.

html_additional_pages
Additional templates that should be rendered to HTML pages, must be a dictionary that maps document names to template names.

Example:

```
html_additional_pages = {
    'download': 'customdownload.html',
}
```

This will render the template customdownload.html as the page download.html.

Note: Earlier versions of Sphinx had a value called html_index which was a clumsy way of controlling the content of the "index" document. If you used this feature, migrate it by adding an 'index' key to this setting, with your custom template as the value, and in your custom template, use

```
{% extend "defindex.html" %}
{% block tables %}
... old template content ...
{% endblock %}
```

html_use_modindex
If true, add a module index to the HTML documents. Default is True.

html_use_index
If true, add an index to the HTML documents. Default is True. New in version 0.4.

html_split_index
If true, the index is generated twice: once as a single page with all the entries, and once as one page per starting letter. Default is False. New in version 0.4.

html_copy_source
If true, the reST sources are included in the HTML build as _sources/name. The default is True.

html_use_opensearch
If nonempty, an *OpenSearch <http://opensearch.org>* description file will be output, and all pages will contain a <link> tag referring to it. Since OpenSearch doesn't support relative URLs for its search page location, the value of this option must be the base URL from which these documents are served (without trailing slash), e.g. "http://docs.python.org". The default is ''.

html_file_suffix
If nonempty, this is the file name suffix for generated HTML files. The default is ".html". New in version 0.4.

html_translator_class
A string with the fully-qualified name of a HTML Translator class, that is, a subclass of Sphinx' HTMLTranslator, that is used to translate document trees to HTML. Default is None (use the builtin translator).

html_show_sphinx
If true, "Created using Sphinx" is shown in the HTML footer. Default is True. New in version 0.4.

htmlhelp_basename
Output file base name for HTML help builder. Default is `'pydoc'`.

6.3 Options for LaTeX output

These options influence LaTeX output.

latex_paper_size
The output paper size (`'letter'` or `'a4'`). Default is `'letter'`.

latex_font_size
The font size ('10pt', '11pt' or '12pt'). Default is `'10pt'`.

latex_documents
This value determines how to group the document tree into LaTeX source files. It must be a list of tuples `(startdocname, targetname, title, author, documentclass, toctree_only)`, where the items are:

- *startdocname*: document name that is the "root" of the LaTeX file. All documents referenced by it in TOC trees will be included in the LaTeX file too. (If you want only one LaTeX file, use your `master_doc` here.)
- *targetname*: file name of the LaTeX file in the output directory.
- *title*: LaTeX document title. Can be empty to use the title of the *startdoc*.
- *author*: Author for the LaTeX document.
- *documentclass*: Must be one of `'manual'` or `'howto'`. Only "manual" documents will get appendices. Also, howtos will have a simpler title page.
- *toctree_only*: Must be `True` or `False`. If `True`, the *startdoc* document itself is not included in the output, only the documents referenced by it via TOC trees. With this option, you can put extra stuff in the master document that shows up in the HTML, but not the LaTeX output.

New in version 0.3: The 6th item `toctree_only`. Tuples with 5 items are still accepted.

latex_logo
If given, this must be the name of an image file (relative to the configuration directory) that is the logo of the docs. It is placed at the top of the title page. Default: `None`.

latex_use_parts
If true, the topmost sectioning unit is parts, else it is chapters. Default: `False`. New in version 0.3.

latex_appendices
A list of document names to append as an appendix to all manuals.

latex_preamble
Additional LaTeX markup for the preamble.

Keep in mind that backslashes must be doubled in Python string literals to avoid interpretation as escape sequences.

latex_use_modindex
If true, add a module index to LaTeX documents. Default is `True`.

CHAPTER

SEVEN

Templating

Sphinx uses the Jinja templating engine for its HTML templates. Jinja is a text-based engine, and inspired by Django templates, so anyone having used Django will already be familiar with it. It also has excellent documentation for those who need to make themselves familiar with it.

7.1 Do I need to use Sphinx' templates to produce HTML?

No. You have several other options:

- You can write a `TemplateBridge` subclass that calls your template engine of choice, and set the `template_bridge` configuration value accordingly.
- You can *write a custom builder* that derives from `StandaloneHTMLBuilder` and calls your template engine of choice.
- You can use the `PickleHTMLBuilder` that produces pickle files with the page contents, and postprocess them using a custom tool, or use them in your Web application.

7.2 Jinja/Sphinx Templating Primer

The default templating language in Sphinx is Jinja. It's Django/Smarty inspired and easy to understand. The most important concept in Jinja is *template inheritance*, which means that you can overwrite only specific blocks within a template, customizing it while also keeping the changes at a minimum.

To customize the output of your documentation you can override all the templates (both the layout templates and the child templates) by adding files with the same name as the original filename into the template directory of the folder the Sphinx quickstart generated for you.

Sphinx will look for templates in the folders of `templates_path` first, and if it can't find the template it's looking for there, it falls back to the builtin templates that come with Sphinx.

A template contains **variables**, which are replaced with values when the template is evaluated, **tags**, which control the logic of the template and **blocks** which are used for template inheritance.

Sphinx provides base templates with a couple of blocks it will fill with data. The default templates are located in the `templates` folder of the Sphinx installation directory. Templates with the same name in the `templates_path` override templates located in the builtin folder.

For example, to add a new link to the template area containing related links all you have to do is to add a new template called `layout.html` with the following contents:

```
{% extends "!layout.html" %}
{% block rootrellink %}
    <li><a href="http://project.invalid/">Project Homepage</a> &raquo;</li>
    {{ super() }}
{% endblock %}
```

By prefixing the name of the extended template with an exclamation mark, Sphinx will load the builtin layout template. If you override a block, you should call `{{ super() }}` somewhere to render the block's content in the extended template – unless you don't want that content to show up.

7.2.1 Blocks

The following blocks exist in the `layout` template:

doctype The doctype of the output format. By default this is XHTML 1.0 Transitional as this is the closest to what Sphinx and Docutils generate and it's a good idea not to change it unless you want to switch to HTML 5 or a different but compatible XHTML doctype.

rellinks This block adds a couple of `<link>` tags to the head section of the template.

extrahead This block is empty by default and can be used to add extra contents into the `<head>` tag of the generated HTML file. This is the right place to add references to JavaScript or extra CSS files.

relbar1 / relbar2 This block contains the list of related links (the parent documents on the left, and the links to index, modules etc. on the right). *relbar1* appears before the document, *relbar2* after the document. By default, both blocks are filled; to show the relbar only before the document, you would override *relbar2* like this:

```
{% block relbar2 %}{% endblock %}
```

rootrellink / relbaritems Inside the relbar there are three sections: The *rootrellink*, the links from the documentation and the *relbaritems*. The *rootrellink* is a block that by default contains a list item pointing to the master document by default, the *relbaritems* is an empty block. If you override them to add extra links into the bar make sure that they are list items and end with the `reldelim1`.

document The contents of the document itself.

sidebar1 / sidebar2 A possible location for a sidebar. *sidebar1* appears before the document and is empty by default, *sidebar2* after the document and contains the default sidebar. If you want to swap the sidebar location override this and call the *sidebar* helper:

```
{% block sidebar1 %}{{ sidebar() }}{% endblock %}
{% block sidebar2 %}{% endblock %}
```

(The *sidebar2* location for the sidebar is needed by the `sphinxdoc.css` stylesheet, for example.)

sidebarlogo The logo location within the sidebar. Override this if you want to place some content at the top of the sidebar.

sidebartoc The table of contents within the sidebar.

sidebarrel The relation links (previous, next document) within the sidebar.

sidebarsearch The search box within the sidebar. Override this if you want to place some content at the bottom of the sidebar.

footer The block for the footer div. If you want a custom footer or markup before or after it, override this one.

7.2.2 Configuration Variables

Inside templates you can set a couple of variables used by the layout template using the `{% set %}` tag:

reldelim1
The delimiter for the items on the left side of the related bar. This defaults to `' &raquo;'` Each item in the related bar ends with the value of this variable.

reldelim2
The delimiter for the items on the right side of the related bar. This defaults to `' |'`. Each item except of the last one in the related bar ends with the value of this variable.

Overriding works like this:

```
{% extends "!layout.html" %}
{% set reldelim1 = ' &gt;' %}
```

7.2.3 Helper Functions

Sphinx provides various Jinja functions as helpers in the template. You can use them to generate links or output multiply used elements.

pathto(*document*)
Return the path to a Sphinx document as a URL. Use this to refer to built documents.

pathto(*file, 1*)
Return the path to a *file* which is a filename relative to the root of the generated output. Use this to refer to static files.

hasdoc(*document*)
Check if a document with the name *document* exists.

sidebar()
Return the rendered sidebar.

relbar()
Return the rendered relbar.

7.2.4 Global Variables

These global variables are available in every template and are safe to use. There are more, but most of them are an implementation detail and might change in the future.

docstitle
The title of the documentation (the value of `html_title`).

sourcename
The name of the copied source file for the current document. This is only nonempty if the `html_copy_source` value is true.

builder
The name of the builder (for builtin builders, `html`, `htmlhelp`, or `web`).

next
The next document for the navigation. This variable is either false or has two attributes *link* and *title*. The title contains HTML markup. For example, to generate a link to the next page, you can use this snippet:

```
{% if next %}
<a href="{{ next.link|e }}">{{ next.title }}</a>
{% endif %}
```

prev
Like `next`, but for the previous page.

CHAPTER

EIGHT

Sphinx Extensions

Since many projects will need special features in their documentation, Sphinx is designed to be extensible on several levels.

First, you can add new *builder*s to support new output formats or actions on the parsed documents. Then, it is possible to register custom reStructuredText roles and directives, extending the markup. And finally, there are so-called "hook points" at strategic places throughout the build process, where an extension can register a hook and run specialized code.

The configuration file itself can be an extension, see the `extensions` configuration value docs.

8.1 Extension API

Each Sphinx extension is a Python module with at least a `setup()` function. This function is called at initialization time with one argument, the application object representing the Sphinx process. This application object has the following public API:

add_builder(*builder*)
Register a new builder. *builder* must be a class that inherits from `Builder`.

add_config_value(*name, default, rebuild_env*)
Register a configuration value. This is necessary for Sphinx to recognize new values and set default values accordingly. The *name* should be prefixed with the extension name, to avoid clashes. The *default* value can be any Python object. The boolean value *rebuild_env* must be `True` if a change in the setting only takes effect when a document is parsed – this means that the whole environment must be rebuilt. Changed in version 0.4: If the *default* value is a callable, it will be called with the config object as its argument in order to get the default value. This can be used to implement config values whose default depends on other values.

add_event(*name*)
Register an event called *name*.

add_node(*node*)
Register a Docutils node class. This is necessary for Docutils internals. It may also be used in the future to validate nodes in the parsed documents.

add_directive(*name, cls, content, arguments, **options*)
Register a Docutils directive. *name* must be the prospective directive name, *func* the directive function for details about the signature and return value. *content*, *arguments* and *options* are set as attributes on the function and determine whether the directive has content, arguments and options, respectively. For their exact meaning, please consult the Docutils documentation.

add_role(*name, role*)
Register a Docutils role. *name* must be the role name that occurs in the source, *role* the role function (see the Docutils documentation on details).

add_description_unit(*directivename, rolename, indextemplate='', parse_node=None, ref_nodeclass=None*)

This method is a very convenient way to add a new type of information that can be cross-referenced. It will do this:

- Create a new directive (called *directivename*) for a *description unit*. It will automatically add index entries if *indextemplate* is nonempty; if given, it must contain exactly one instance of `%s`. See the example below for how the template will be interpreted.
- Create a new role (called *rolename*) to cross-reference to these description units.
- If you provide *parse_node*, it must be a function that takes a string and a docutils node, and it must populate the node with children parsed from the string. It must then return the name of the item to be used in cross-referencing and index entries. See the `ext.py` file in the source for this documentation for an example.

For example, if you have this call in a custom Sphinx extension:

```
app.add_description_unit('directive', 'dir', 'pair: %s; directive')
```

you can use this markup in your documents:

```
.. directive:: function

   Document a function.

<...>

See also the :dir:`function` directive.
```

For the directive, an index entry will be generated as if you had prepended

```
.. index:: pair: function; directive
```

The reference node will be of class `literal` (so it will be rendered in a proportional font, as appropriate for code) unless you give the *ref_nodeclass* argument, which must be a docutils node class (most useful are `docutils.nodes.emphasis` or `docutils.nodes.strong` – you can also use `docutils.nodes.generated` if you want no further text decoration).

For the role content, you have the same options as for standard Sphinx roles (see *Cross-referencing syntax*).

add_crossref_type(*directivename, rolename, indextemplate='', ref_nodeclass=None*)

This method is very similar to `add_description_unit()` except that the directive it generates must be empty, and will produce no output.

That means that you can add semantic targets to your sources, and refer to them using custom roles instead of generic ones (like `ref`). Example call:

```
app.add_crossref_type('topic', 'topic', 'single: %s', docutils.nodes.emphasis)
```

Example usage:

```
.. topic:: application API

The application API
-------------------

<...>

See also :topic:`this section <application API>`.
```

(Of course, the element following the `topic` directive needn't be a section.)

add_transform(*transform*)
Add the standard docutils `Transform` subclass *transform* to the list of transforms that are applied after Sphinx parses a reST document.

connect(*event, callback*)
Register *callback* to be called when *event* is emitted. For details on available core events and the arguments of callback functions, please see *Sphinx core events*.

The method returns a "listener ID" that can be used as an argument to `disconnect()`.

disconnect(*listener_id*)
Unregister callback *listener_id*.

emit(*event, *arguments*)
Emit *event* and pass *arguments* to the callback functions. Do not emit core Sphinx events in extensions!

exception ExtensionError
All these functions raise this exception if something went wrong with the extension API.

Examples of using the Sphinx extension API can be seen in the `sphinx.ext` package.

8.1.1 Sphinx core events

These events are known to the core. The arguments shown are given to the registered event handlers.

builder-initedapp
Emitted the builder object has been created.

doctree-readapp, doctree
Emitted when a doctree has been parsed and read by the environment, and is about to be pickled.

doctree-resolvedapp, doctree, docname
Emitted when a doctree has been "resolved" by the environment, that is, all references and TOCs have been inserted.

page-contextapp, pagename, templatename, context, doctree
Emitted when the HTML builder has created a context dictionary to render a template with – this can be used to add custom elements to the context.

The *pagename* argument is the canonical name of the page being rendered, that is, without `.html` suffix and using slashes as path separators. The *templatename* is the name of the template to render, this will be `'page.html'` for all pages from reST documents.

The *context* argument is a dictionary of values that are given to the template engine to render the page and can be modified to include custom values. Keys must be strings.

The *doctree* argument will be a doctree when the page is created from a reST documents; it will be `None` when the page is created from an HTML template alone. New in version 0.4.

8.1.2 The template bridge

class TemplateBridge()
This class defines the interface for a "template bridge", that is, a class that renders templates given a template name and a context.

init(*builder*)
Called by the builder to initialize the template system. *builder* is the builder object; you'll probably want to look at the value of `builder.config.templates_path`.

newest_template_mtime()
Called by the builder to determine if output files are outdated because of template changes. Return the mtime of the newest template file that was changed. The default implementation returns `0`.

render(*template, context*)
Called by the builder to render a *template* with a specified context (a Python dictionary).

8.2 Writing new builders

XXX to be expanded.

class Builder()
This is the base class for all builders.

These methods are predefined and will be called from the application:

load_env()
Set up the build environment.

get_relative_uri(*from_, to, typ=None*)
Return a relative URI between two source filenames. May raise environment.NoUri if there's no way to return a sensible URI.

build_all()
Build all source files.

build_specific(*filenames*)
Only rebuild as much as needed for changes in the source_filenames.

build_update()
Only rebuild files changed or added since last build.

build(*docnames, summary=None, method='update'*)

These methods must be overridden in concrete builder classes:

init()
Load necessary templates and perform initialization.

get_outdated_docs()
Return an iterable of output files that are outdated, or a string describing what an update build will build.

get_target_uri(*docname, typ=None*)
Return the target URI for a document name (typ can be used to qualify the link characteristic for individual builders).

prepare_writing(*docnames*)

write_doc(*docname, doctree*)

finish()

8.3 Builtin Sphinx extensions

These extensions are built in and can be activated by respective entries in the `extensions` configuration value:

8.3.1 `sphinx.ext.autodoc` – Include documentation from docstrings

This extension can import the modules you are documenting, and pull in documentation from docstrings in a semi-automatic way.

For this to work, the docstrings must of course be written in correct reStructuredText. You can then use all of the usual Sphinx markup in the docstrings, and it will end up correctly in the documentation. Together with hand-written documentation, this technique eases the pain of having to maintain two locations for documentation, while at the same time avoiding auto-generated-looking pure API documentation.

autodoc provides several directives that are versions of the usual module, class and so forth. On parsing time, they import the corresponding module and extract the docstring of the given objects, inserting them into the page source under a suitable module, class etc. directive.

Note: Just as class respects the current module, autoclass will also do so, and likewise with method and class.

.. **automodule::**
.. **autoclass::**
.. **autoexception::**
Document a module, class or exception. All three directives will by default only insert the docstring of the object itself:

```
.. autoclass:: Noodle
```

will produce source like this:

```
.. class:: Noodle

   Noodle's docstring.
```

The "auto" directives can also contain content of their own, it will be inserted into the resulting non-auto directive source after the docstring (but before any automatic member documentation).

Therefore, you can also mix automatic and non-automatic member documentation, like so:

```
.. autoclass:: Noodle
   :members: eat, slurp

   .. method:: boil(time=10)

      Boil the noodle *time* minutes.
```

Options and advanced usage

- If you want to automatically document members, there's a members option:

  ```
  .. autoclass:: Noodle
     :members:
  ```

  will document all non-private member functions and properties (that is, those whose name doesn't start with _), while

  ```
  .. autoclass:: Noodle
     :members: eat, slurp
  ```

  will document exactly the specified members.

- Members without docstrings will be left out, unless you give the undoc-members flag option:

  ```
  .. autoclass:: Noodle
     :members:
     :undoc-members:
  ```

- For classes and exceptions, members inherited from base classes will be left out, unless you give the inherited-members flag option, in addition to members:

  ```
  .. autoclass:: Noodle
     :members:
     :inherited-members:
  ```

  This can be combined with undoc-members to document *all* available members of the class or module. New in version 0.3.

- It's possible to override the signature for callable members (functions, methods, classes) with the regular syntax that will override the signature gained from instropection:

  ```
  .. autoclass:: Noodle(type)

     .. automethod:: eat(persona)
  ```

  This is useful if the signature from the method is hidden by a decorator. New in version 0.4.
- The `autoclass` and `autoexception` directives also support a flag option called `show-inheritance`. When given, a list of base classes will be inserted just below the class signature. New in version 0.4.
- All autodoc directives support the `noindex` flag option that has the same effect as for standard `function` etc. directives: no index entries are generated for the documented object (and all autodocumented members). New in version 0.4.

Note: In an `automodule` directive with the `members` option set, only module members whose `__module__` attribute is equal to the module name as given to `automodule` will be documented. This is to prevent documentation of imported classes or functions.

.. autofunction::
.. automethod::
.. autoattribute::
These work exactly like `autoclass` etc., but do not offer the options used for automatic member documentation.

Note: If you document decorated functions or methods, keep in mind that autodoc retrieves its docstrings by importing the module and inspecting the `__doc__` attribute of the given function or method. That means that if a decorator replaces the decorated function with another, it must copy the original `__doc__` to the new function.

From Python 2.5, `functools.wraps()` can be used to create well-behaved decorating functions.

There are also new config values that you can set:

automodule_skip_lines
This value (whose default is `0`) can be used to skip an amount of lines in every module docstring that is processed by an `automodule` directive. This is provided because some projects like to put headings in the module docstring, which would then interfere with your sectioning, or automatic fields with version control tags, that you don't want to put in the generated documentation. **Deprecated since release 0.4.** Use the more versatile docstring processing provided by `autodoc-process-docstring`.

autoclass_content
This value selects what content will be inserted into the main body of an `autoclass` directive. The possible values are:

"class" Only the class' docstring is inserted. This is the default. You can still document `__init__` as a separate method using `automethod` or the `members` option to `autoclass`.

"both" Both the class' and the `__init__` method's docstring are concatenated and inserted.

"init" Only the `__init__` method's docstring is inserted.

New in version 0.3.

Docstring preprocessing

New in version 0.4. autodoc provides the following additional event:

autodoc-process-docstringapp, what, name, obj, options, lines
Emitted when autodoc has read and processed a docstring. *lines* is a list of strings – the lines of the processed docstring – that the event handler can modify **in place** to change what Sphinx puts into the output.

Parameters
- *app* – the Sphinx application object
- *what* – the type of the object which the docstring belongs to (one of `"module"`, `"class"`, `"exception"`, `"function"`, `"method"`, `"attribute"`)
- *name* – the fully qualified name of the object
- *obj* – the object itself
- *options* – the options given to the directive: an object with attributes `inherited_members`, `undoc_members`, `show_inheritance` and `noindex` that are true if the flag option of same name was given to the auto directive
- *lines* – the lines of the docstring, see above

The `sphinx.ext.autodoc` module provides factory functions for commonly needed docstring processing:

`cut_lines`(*pre, post=0, what=None*)
Return a listener that removes the first *pre* and last *post* lines of every docstring. If *what* is a sequence of strings, only docstrings of a type in *what* will be processed.

Use like this (e.g. in the `setup()` function of `conf.py`):

```
from sphinx.ext.autodoc import cut_lines
app.connect('autodoc-process-docstring', cut_lines(4, what=['module']))
```

This can (and should) be used in place of `automodule_skip_lines`.

`between`(*marker, what=None, keepempty=False*)
Return a listener that only keeps lines between lines that match the *marker* regular expression. If no line matches, the resulting docstring would be empty, so no change will be made unless *keepempty* is true.

If *what* is a sequence of strings, only docstrings of a type in *what* will be processed.

8.3.2 `sphinx.ext.doctest` – Test snippets in the documentation

This extension allows you to test snippets in the documentation in a natural way. It works by collecting specially-marked up code blocks and running them as doctest tests.

Within one document, test code is partitioned in *groups*, where each group consists of:

- zero or more *setup code* blocks (e.g. importing the module to test)
- one or more *test* blocks

When building the docs with the `doctest` builder, groups are collected for each document and run one after the other, first executing setup code blocks, then the test blocks in the order they appear in the file.

There are two kinds of test blocks:

- *doctest-style* blocks mimic interactive sessions by interleaving Python code (including the interpreter prompt) and output.
- *code-output-style* blocks consist of an ordinary piece of Python code, and optionally, a piece of output for that code.

The doctest extension provides four directives. The *group* argument is interpreted as follows: if it is empty, the block is assigned to the group named `default`. If it is `*`, the block is assigned to all groups (including the `default` group). Otherwise, it must be a comma-separated list of group names.

`.. testsetup:: [group]`
A setup code block. This code is not shown in the output for other builders, but executed before the doctests of the group(s) it belongs to.

.. **doctest:: [group]**
A doctest-style code block. You can use standard `doctest` flags for controlling how actual output is compared with what you give as output. By default, these options are enabled: `ELLIPSIS` (allowing you to put ellipses in the expected output that match anything in the actual output), `IGNORE_EXCEPTION_DETAIL` (not comparing tracebacks), `DONT_ACCEPT_TRUE_FOR_1` (by default, doctest accepts "True" in the output where "1" is given – this is a relic of pre-Python 2.2 times).

This directive supports two options:

- `hide`, a flag option, hides the doctest block in other builders. By default it is shown as a highlighted doctest block.
- `options`, a string option, can be used to give a comma-separated list of doctest flags that apply to each example in the tests. (You still can give explicit flags per example, with doctest comments, but they will show up in other builders too.)

Note that like with standard doctests, you have to use `<BLANKLINE>` to signal a blank line in the expected output. The `<BLANKLINE>` is removed when building presentation output (HTML, LaTeX etc.).

Also, you can give inline doctest options, like in doctest:

```
>>> datetime.date.now()   # doctest: +SKIP
datetime.date(2008, 1, 1)
```

They will be respected when the test is run, but stripped from presentation output.

.. **testcode:: [group]**
A code block for a code-output-style test.

This directive supports one option:

- `hide`, a flag option, hides the code block in other builders. By default it is shown as a highlighted code block.

.. **testoutput:: [group]**
The corresponding output for the last `testcode` block.

This directive supports two options:

- `hide`, a flag option, hides the output block in other builders. By default it is shown as a literal block without highlighting.
- `options`, a string option, can be used to give doctest flags (comma-separated) just like in normal doctest blocks.

Example:

```
.. testoutput::
   :hide:
   :options: -ELLIPSIS, +NORMALIZE_WHITESPACE

   Output text.
```

The following is an example for the usage of the directives. The test via `doctest` and the test via `testcode` and `testoutput` are completely equivalent.

```
The parrot module
=================

.. testsetup:: *

   import parrot
```

```
The parrot module is a module about parrots.

Doctest example:

.. doctest::

   >>> parrot.voom(3000)
   This parrot wouldn't voom if you put 3000 volts through it!

Test-Output example:

.. testcode::

   parrot.voom(3000)

This would output:

.. testoutput::

   This parrot wouldn't voom if you put 3000 volts through it!
```

There are also these config values for customizing the doctest extension:

doctest_path
A list of directories that will be added to `sys.path` when the doctest builder is used. (Make sure it contains absolute paths.)

doctest_test_doctest_blocks
If this is a nonempty string (the default is `'default'`), standard reST doctest blocks will be tested too. They will be assigned to the group name given.

reST doctest blocks are simply doctests put into a paragraph of their own, like so:

```
Some documentation text.

>>> print 1
1

Some more documentation text.
```

(Note that no special `::` is needed to introduce the block; docutils recognizes it from the leading `>>>`. Also, no additional indentation is necessary, though it doesn't hurt.)

If this value is left at its default value, the above snippet is interpreted by the doctest builder exactly like the following:

```
Some documentation text.

.. doctest::

   >>> print 1
   1

Some more documentation text.
```

This feature makes it easy for you to test doctests in docstrings included with the `autodoc` extension without marking them up with a special directive.

Note though that you can't have blank lines in reST doctest blocks. They will be interpreted as one block ending and another one starting. Also, removal of `<BLANKLINE>` and `# doctest:` options only works in `doctest` blocks.

8.3.3 `sphinx.ext.refcounting` – Keep track of reference counting behavior

XXX to be written.

8.3.4 `sphinx.ext.ifconfig` – Include content based on configuration

This extension is quite simple, and features only one directive:

`..` **ifconfig::**
Include content of the directive only if the Python expression given as an argument is `True`, evaluated in the namespace of the project's configuration (that is, all variables from `conf.py` are available).

For example, one could write

```
.. ifconfig:: releaselevel in ('alpha', 'beta', 'rc')

   This stuff is only included in the built docs for unstable versions.
```

8.3.5 `sphinx.ext.coverage` – Collect doc coverage stats

This extension features one additional builder, the `CoverageBuilder`.

class CoverageBuilder()
To use this builder, activate the coverage extension in your configuration file and give `-b coverage` on the command line.

XXX to be expanded.

Several new configuration values can be used to specify what the builder should check:

coverage_ignore_modules

coverage_ignore_functions

coverage_ignore_classes

coverage_c_path

coverage_c_regexes

coverage_ignore_c_items

CHAPTER

NINE

Glossary

builder A class (inheriting from `Builder`) that takes parsed documents and performs an action on them. Normally, builders translate the documents to an output format, but it is also possible to use the builder builders that e.g. check for broken links in the documentation, or build coverage information.

See *Available builders* for an overview over Sphinx' built-in builders.

configuration directory The directory containing `conf.py`. By default, this is the same as the *source directory*, but can be set differently with the **-c** command-line option.

description unit The basic building block of Sphinx documentation. Every "description directive" (e.g. `function` or `describe`) creates such a unit; and most units can be cross-referenced to.

environment A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

source directory The directory which, including its subdirectories, contains all source files for one Sphinx project.

CHAPTER

TEN

Changes in Sphinx

10.1 Release 0.4.2 (Jul 29, 2008)

- Fix rendering of the `samp` role in HTML.
- Fix a bug with LaTeX links to headings leading to a wrong page.
- Reread documents with globbed toctrees when source files are added or removed.
- Add a missing parameter to PickleHTMLBuilder.handle_page().
- Put inheritance info always on its own line.
- Don't automatically enclose code with whitespace in it in quotes; only do this for the `samp` role.
- autodoc now emits a more precise error message when a module can't be imported or an attribute can't be found.
- The JavaScript search now uses the correct file name suffix when referring to found items.
- The automodule directive now accepts the `inherited-members` and `show-inheritance` options again.
- You can now rebuild the docs normally after relocating the source and/or doctree directory.

10.2 Release 0.4.1 (Jul 5, 2008)

- Added sub-/superscript node handling to TextBuilder.
- Label names in references are now case-insensitive, since reST label names are always lowercased.
- Fix linkcheck builder crash for malformed URLs.
- Add compatibility for admonitions and docutils 0.5.
- Remove the silly restriction on "rubric" in the LaTeX writer: you can now write arbitrary "rubric" directives, and only those with a title of "Footnotes" will be ignored.
- Copy the HTML logo to the output `_static` directory.
- Fix LaTeX code for modules with underscores in names and platforms.
- Fix a crash with nonlocal image URIs.
- Allow the usage of :noindex: in `automodule` directives, as documented.
- Fix the `delete()` docstring processor function in autodoc.

- Fix warning message for nonexisting images.
- Fix JavaScript search in Internet Explorer.

10.3 Release 0.4 (Jun 23, 2008)

10.3.1 New features added

- `tocdepth` can be given as a file-wide metadata entry, and specifies the maximum depth of a TOC of this file.
- The new config value *default_role* can be used to select the default role for all documents.
- Sphinx now interprets field lists with fields like `:param foo:` in description units.
- The new *staticmethod* directive can be used to mark methods as static methods.
- HTML output:
  - The "previous" and "next" links have a more logical structure, so that by following "next" links you can traverse the entire TOC tree.
  - The new event *html-page-context* can be used to include custom values into the context used when rendering an HTML template.
  - Document metadata is now in the default template context, under the name *metadata*.
  - The new config value *html_favicon* can be used to set a favicon for the HTML output. Thanks to Sebastian Wiesner.
  - The new config value *html_use_index* can be used to switch index generation in HTML documents off.
  - The new config value *html_split_index* can be used to create separate index pages for each letter, to be used when the complete index is too large for one page.
  - The new config value *html_short_title* can be used to set a shorter title for the documentation which is then used in the navigation bar.
  - The new config value *html_show_sphinx* can be used to control whether a link to Sphinx is added to the HTML footer.
  - The new config value *html_file_suffix* can be used to set the HTML file suffix to e.g. `.xhtml`.
  - The directories in the *html_static_path* can now contain subdirectories.
  - The module index now isn't collapsed if the number of submodules is larger than the number of toplevel modules.
- The image directive now supports specifying the extension as `.*`, which makes the builder select the one that matches best. Thanks to Sebastian Wiesner.
- The new config value *exclude_trees* can be used to exclude whole subtrees from the search for source files.
- Defaults for configuration values can now be callables, which allows dynamic defaults.
- The new TextBuilder creates plain-text output.
- Python 3-style signatures, giving a return annotation via `->`, are now supported.
- Extensions:
  - The autodoc extension now offers a much more flexible way to manipulate docstrings before including them into the output, via the new *autodoc-process-docstring* event.
  - The *autodoc* extension accepts signatures for functions, methods and classes now that override the signature got via introspection from Python code.

  - The *autodoc* extension now offers a `show-inheritance` option for autoclass that inserts a list of bases after the signature.
  - The autodoc directives now support the `noindex` flag option.

10.3.2 Bugs fixed

- Correctly report the source location for docstrings included with autodoc.
- Fix the LaTeX output of description units with multiple signatures.
- Handle the figure directive in LaTeX output.
- Handle raw admonitions in LaTeX output.
- Fix determination of the title in HTML help output.
- Handle project names containing spaces.
- Don't write SSI-like comments in HTML output.
- Rename the "sidebar" class to "sphinxsidebar" in order to stay different from reST sidebars.
- Use a binary TOC in HTML help generation to fix issues links without explicit anchors.
- Fix behavior of references to functions/methods with an explicit title.
- Support citation, subscript and superscript nodes in LaTeX writer.
- Provide the standard "class" directive as "cssclass"; else it is shadowed by the Sphinx-defined directive.
- Fix the handling of explicit module names given to autoclass directives. They now show up with the correct module name in the generated docs.
- Enable autodoc to process Unicode docstrings.
- The LaTeX writer now translates line blocks with `\raggedright`, which plays nicer with tables.
- Fix bug with directories in the HTML builder static path.

10.4 Release 0.3 (May 6, 2008)

10.4.1 New features added

- The `toctree` directive now supports a `glob` option that allows glob-style entries in the content.
- If the *pygments_style* config value contains a dot it's treated as the import path of a custom Pygments style class.
- A new config value, *exclude_dirs*, can be used to exclude whole directories from the search for source files.
- The configuration directory (containing `conf.py`) can now be set independently from the source directory. For that, a new command-line option `-c` has been added.
- A new directive `tabularcolumns` can be used to give a tabular column specification for LaTeX output. Tables now use the `tabulary` package. Literal blocks can now be placed in tables, with several caveats.
- A new config value, *latex_use_parts*, can be used to enable parts in LaTeX documents.
- Autodoc now skips inherited members for classes, unless you give the new `inherited-members` option.

- A new config value, *autoclass_content*, selects if the docstring of the class' `__init__` method is added to the directive's body.
- Support for C++ class names (in the style `Class::Function`) in C function descriptions.
- Support for a `toctree_only` item in items for the `latex_documents` config value. This only includes the documents referenced by TOC trees in the output, not the rest of the file containing the directive.

10.4.2 Bugs fixed

- sphinx.htmlwriter: Correctly write the TOC file for any structure of the master document. Also encode non-ASCII characters as entities in TOC and index file. Remove two remaining instances of hard-coded "documentation".
- sphinx.ext.autodoc: descriptors are detected properly now.
- sphinx.latexwriter: implement all reST admonitions, not just `note` and `warning`.
- Lots of little fixes to the LaTeX output and style.
- Fix OpenSearch template and make template URL absolute. The *html_use_opensearch* config value now must give the base URL.
- Some unused files are now stripped from the HTML help file build.

10.5 Release 0.2 (Apr 27, 2008)

10.5.1 Incompatible changes

- Jinja, the template engine used for the default HTML templates, is now no longer shipped with Sphinx. If it is not installed automatically for you (it is now listed as a dependency in `setup.py`), install it manually from PyPI. This will also be needed if you're using Sphinx from a SVN checkout; in that case please also remove the `sphinx/jinja` directory that may be left over from old revisions.
- The clumsy handling of the `index.html` template was removed. The config value `html_index` is gone, and `html_additional_pages` should be used instead. If you need it, the old `index.html` template is still there, called `defindex.html`, and you can port your html_index template, using Jinja inheritance, by changing your template:

  ```
  {% extends "defindex.html" %}
  {% block tables %}
  ... old html_index template content ...
  {% endblock %}
  ```

  and putting `'index': name of your template` in `html_additional_pages`.
- In the layout template, redundant `blocks` were removed; you should use Jinja's standard `{{ super() }}` mechanism instead, as explained in the (newly written) templating docs.

10.5.2 New features added

- Extension API (Application object):
  - Support a new method, `add_crossref_type`. It works like `add_description_unit` but the directive will only create a target and no output.

  - Support a new method, `add_transform`. It takes a standard docutils `Transform` subclass which is then applied by Sphinx' reader on parsing reST document trees.
  - Add support for other template engines than Jinja, by adding an abstraction called a "template bridge". This class handles rendering of templates and can be changed using the new configuration value "template_bridge".
  - The config file itself can be an extension (if it provides a `setup()` function).

- Markup:
  - New directive, `currentmodule`. It can be used to indicate the module name of the following documented things without creating index entries.
  - Allow giving a different title to documents in the toctree.
  - Allow giving multiple options in a `cmdoption` directive.
  - Fix display of class members without explicit class name given.

- Templates (HTML output):
  - `index.html` renamed to `defindex.html`, see above.
  - There's a new config value, `html_title`, that controls the overall "title" of the set of Sphinx docs. It is used instead everywhere instead of "Projectname vX.Y documentation" now.
  - All references to "documentation" in the templates have been removed, so that it is now easier to use Sphinx for non-documentation documents with the default templates.
  - Templates now have an XHTML doctype, to be consistent with docutils' HTML output.
  - You can now create an OpenSearch description file with the `html_use_opensearch` config value.
  - You can now quickly include a logo in the sidebar, using the `html_logo` config value.
  - There are new blocks in the sidebar, so that you can easily insert content into the sidebar.

- LaTeX output:
  - The `sphinx.sty` package was cleaned of unused stuff.
  - You can include a logo in the title page with the `latex_logo` config value.
  - You can define the link colors and a border and background color for verbatim environments.

Thanks to Jacob Kaplan-Moss, Talin, Jeroen Ruigrok van der Werven and Sebastian Wiesner for suggestions.

10.5.3 Bugs fixed

- sphinx.ext.autodoc: Don't check `__module__` for explicitly given members. Remove "self" in class constructor argument list.
- sphinx.htmlwriter: Don't use os.path for joining image HREFs.
- sphinx.htmlwriter: Don't use SmartyPants for HTML attribute values.
- sphinx.latexwriter: Implement option lists. Also, some other changes were made to `sphinx.sty` in order to enhance compatibility and remove old unused stuff. Thanks to Gael Varoquaux for that!
- sphinx.roles: Fix referencing glossary terms with explicit targets.
- sphinx.environment: Don't swallow TOC entries when resolving subtrees.
- sphinx.quickstart: Create a sensible default latex_documents setting.
- sphinx.builder, sphinx.environment: Gracefully handle some user error cases.
- sphinx.util: Follow symbolic links when searching for documents.

10.6 Release 0.1.61950 (Mar 26, 2008)

- sphinx.quickstart: Fix format string for Makefile.

10.7 Release 0.1.61945 (Mar 26, 2008)

- sphinx.htmlwriter, sphinx.latexwriter: Support the `.. image::` directive by copying image files to the output directory.
- sphinx.builder: Consistently name "special" HTML output directories with a leading underscore; this means `_sources` and `_static`.
- sphinx.environment: Take dependent files into account when collecting the set of outdated sources.
- sphinx.directives: Record files included with `.. literalinclude::` as dependencies.
- sphinx.ext.autodoc: Record files from which docstrings are included as dependencies.
- sphinx.builder: Rebuild all HTML files in case of a template change.
- sphinx.builder: Handle unavailability of TOC relations (previous/ next chapter) more gracefully in the HTML builder.
- sphinx.latexwriter: Include fncychap.sty which doesn't seem to be very common in TeX distributions. Add a `clean` target in the latex Makefile. Really pass the correct paper and size options to the LaTeX document class.
- setup: On Python 2.4, don't egg-depend on docutils if a docutils is already installed – else it will be overwritten.

10.8 Release 0.1.61843 (Mar 24, 2008)

- sphinx.quickstart: Really don't create a makefile if the user doesn't want one.
- setup: Don't install scripts twice, via setuptools entry points and distutils scripts. Only install via entry points.
- sphinx.builder: Don't recognize the HTML builder's copied source files (under `_sources`) as input files if the source suffix is `.txt`.
- sphinx.highlighting: Generate correct markup for LaTeX Verbatim environment escapes even if Pygments is not installed.
- sphinx.builder: The WebHTMLBuilder is now called PickleHTMLBuilder.
- sphinx.htmlwriter: Make parsed-literal blocks work as expected, not highlighting them via Pygments.
- sphinx.environment: Don't error out on reading an empty source file.

10.9 Release 0.1.61798 (Mar 23, 2008)

- sphinx: Work with docutils SVN snapshots as well as 0.4.
- sphinx.ext.doctest: Make the group in which doctest blocks are placed selectable, and default to `'default'`.
- sphinx.ext.doctest: Replace `<BLANKLINE>` in doctest blocks by real blank lines for presentation output, and remove doctest options given inline.

- sphinx.environment: Move doctest_blocks out of block_quotes to support indented doctest blocks.
- sphinx.ext.autodoc: Render `.. automodule::` docstrings in a section node, so that module docstrings can contain proper sectioning.
- sphinx.ext.autodoc: Use the module's encoding for decoding docstrings, rather than requiring ASCII.

10.10 Release 0.1.61611 (Mar 21, 2008)

- First public release.

CHAPTER

ELEVEN

Projects using Sphinx

This is an (incomplete) list of projects that use Sphinx or are experimenting with using it for their documentation. If you like to be included, please mail to sphinx-dev@googlegroups.com.

- Sphinx: http://sphinx.pocoo.org/
- Python: http://docs.python.org/dev/
- NumPy: http://mentat.za.net/numpy/refguide/
- Pylons: http://bel-epa.com/pylonsdocs/
- Jinja: http://jinja.pocoo.org/2/documentation/
- F2py: http://www.f2py.org/html/
- Paver: http://www.blueskyonmars.com/projects/paver/
- Satchmo: http://www.satchmoproject.com/docs/svn/
- PyEphem: http://rhodesmill.org/pyephem/
- Paste: http://pythonpaste.org/script/
- Calibre: http://calibre.kovidgoyal.net/user_manual/
- PyUblas: http://tiker.net/doc/pyublas/
- Py on Windows: http://timgolden.me.uk/python-on-windows/
- Zope 3: e.g. http://docs.carduner.net/z3c-tutorial/
- SymPy: http://docs.sympy.org/
- Grok (upcoming)
- Django (upcoming)
- Matplotlib (upcoming)
- TurboGears (upcoming)

MODULE INDEX

C

S

INDEX

R

S

T

W