Sphinx Documentation

Release 1.0.8

Georg Brandl

November 21, 2012

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. That is, if you have a directory containing a bunch of reST-formatted documents (and possibly subdirectories of docs in there as well), Sphinx can generate a nicely-organized arrangement of HTML files (in some other directory) for easy browsing and navigation. But from the same source, it can also generate a LaTeX file that you can compile into a PDF version of the documents, or a PDF file directly using rst2pdf.

The focus is on hand-written documentation, rather than auto-generated API docs. Though there is 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 Conversion from other systems

This section is intended to collect helpful hints for those wanting to migrate to reStructuredText/Sphinx from other documentation systems.

- Gerard Flanagan has written a script to convert pure HTML to reST; it can be found at BitBucket.
- For converting the old Python docs to Sphinx, a converter was written which can be found at the Python SVN repository. It contains generic code to convert Python-doc-style LaTeX markup to Sphinx reST.
- Marcin Wojdyr has written a script to convert Docbook to reST with Sphinx markup; it is at Google Code.
- Christophe de Vienne wrote a tool to convert from Open/LibreOffice documents to Sphinx: odt2sphinx.
- To convert different markups, Pandoc is a very helpful tool.

1.2 Use with other systems

See the *pertinent section in the FAQ list*.

1.3 Prerequisites

Sphinx needs at least **Python 2.4** to run, as well as the docutils and Jinja2 libraries. Sphinx should work with docutils version 0.5 or some (not broken) SVN trunk snapshot. If you like to have source code highlighting

support, you must also install the Pygments library.

1.4 Usage

See *First Steps with Sphinx* for an introduction. It also contains links to more advanced sections in this manual for the topics it discusses.

CHAPTER

TWO

FIRST STEPS WITH SPHINX

This document is meant to give a tutorial-like overview of all common tasks while using Sphinx.

The green arrows designate "more info" links leading to advanced sections about the described task.

2.1 Setting up the documentation sources

The root directory of a documentation collection is called the *source directory*. This directory also contains the Sphinx configuration file `conf.py`, where you can configure all aspects of how Sphinx reads your sources and builds your documentation. [1]

Sphinx comes with a script called **sphinx-quickstart** that sets up a source directory and creates a default `conf.py` with the most useful configuration values from a few questions it asks you. Just run

```
$ sphinx-quickstart
```

and answer its questions. (Be sure to say yes to the "autodoc" extension.)

2.2 Defining document structure

Let's assume you've run **sphinx-quickstart**. It created a source directory with `conf.py` and a master document, `index.rst` (if you accepted the defaults). The main function of the *master document* is to serve as a welcome page, and to contain the root of the "table of contents tree" (or *toctree*). This is one of the main things that Sphinx adds to reStructuredText, a way to connect multiple files to a single hierarchy of documents.

[1] This is the usual lay-out. However, `conf.py` can also live in another directory, the *configuration directory*. See *Invocation of sphinx-build*.

reStructuredText directives

`toctree` is a reStructuredText *directive*, a very versatile piece of markup. Directives can have arguments, options and content.
Arguments are given directly after the double colon following the directive's name. Each directive decides whether it can have arguments, and how many.
Options are given after the arguments, in form of a "field list". The `maxdepth` is such an option for the `toctree` directive.
Content follows the options or arguments after a blank line. Each directive decides whether to allow content, and what to do with it.
A common gotcha with directives is that **the first line of the content must be indented to the same level as the options are**.

The toctree directive initially is empty, and looks like this:

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

You add documents listing them in the *content* of the directive:

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

   intro
   tutorial
   ...
```

This is exactly how the toctree for this documentation looks. The documents to include are given as *document name*s, which in short means that you leave off the file name extension and use slashes as directory separators.


Read more about *the toctree directive*.

You can now create the files you listed in the toctree and add content, and their section titles will be inserted (up to the "maxdepth" level) at the place where the toctree directive is placed. Also, Sphinx now knows about the order and hierarchy of your documents. (They may contain `toctree` directives themselves, which means you can create deeply nested hierarchies if necessary.)

2.3 Adding content

In Sphinx source files, you can use most features of standard reStructuredText. There are also several features added by Sphinx. For example, you can add cross-file references in a portable way (which works for all output types) using the `ref` role.

For an example, if you are viewing the HTML version you can look at the source for this document – use the "Show Source" link in the sidebar.


See *reStructuredText Primer* for a more in-depth introduction to reStructuredText and *Sphinx Markup Constructs* for a full list of markup added by Sphinx.

2.4 Running the build

Now that you have added some files and content, let's make a first build of the docs. A build is started with the **sphinx-build** program, called like this:

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

where *sourcedir* is the *source directory*, and *builddir* is the directory in which you want to place the built documentation. The *-b* option selects a builder; in this example Sphinx will build HTML files.

See *Invocation of sphinx-build* for all options that **sphinx-build** supports.

However, **sphinx-quickstart** script creates a `Makefile` and a `make.bat` which make life even easier for you: with them you only need to run

```
$ make html
```

to build HTML docs in the build directory you chose. Execute `make` without an argument to see which targets are available.

2.5 Documenting objects

One of Sphinx' main objectives is easy documentation of *objects* (in a very general sense) in any *domain*. A domain is a collection of object types that belong together, complete with markup to create and reference descriptions of these objects.

The most prominent domain is the Python domain. To e.g. document the Python built-in function `enumerate()`, you would add this to one of your source files:

```
.. py:function:: enumerate(sequence[, start=0])

   Return an iterator that yields tuples of an index and an item of the
   *sequence*. (And so on.)
```

This is rendered like this:

enumerate(*sequence*[, *start=0*])
Return an iterator that yields tuples of an index and an item of the *sequence*. (And so on.)

The argument of the directive is the *signature* of the object you describe, the content is the documentation for it. Multiple signatures can be given, each in its own line.

The Python domain also happens to be the default domain, so you don't need to prefix the markup with the domain name:

```
.. function:: enumerate(sequence[, start=0])

   ...
```

does the same job if you keep the default setting for the default domain.

There are several more directives for documenting other types of Python objects, for example `py:class` or `py:method`. There is also a cross-referencing *role* for each of these object types. This markup will create a link to the documentation of `enumerate()`:

```
The :py:func:`enumerate` function can be used for ...
```

And here is the proof: A link to `enumerate()`.

Again, the `py:` can be left out if the Python domain is the default one. It doesn't matter which file contains the actual documentation for `enumerate()`; Sphinx will find it and create a link to it.

Each domain will have special rules for how the signatures can look like, and make the formatted output look pretty, or add specific features like links to parameter types, e.g. in the C/C++ domains.

See *Sphinx Domains* for all the available domains and their directives/roles.

2.6 Basic configuration

Earlier we mentioned that the `conf.py` file controls how Sphinx processes your documents. In that file, which is executed as a Python source file, you assign configuration values. For advanced users: since it is executed by Sphinx, you can do non-trivial tasks in it, like extending `sys.path` or importing a module to find out the version your are documenting.

The config values that you probably want to change are already put into the `conf.py` by **sphinx-quickstart** and initially commented out (with standard Python syntax: a `#` comments the rest of the line). To change the default value, remove the hash sign and modify the value. To customize a config value that is not automatically added by **sphinx-quickstart**, just add an additional assignment.

Keep in mind that the file uses Python syntax for strings, numbers, lists and so on. The file is saved in UTF-8 by default, as indicated by the encoding declaration in the first line. If you use non-ASCII characters in any string value, you need to use Python Unicode strings (like `project = u'Exposé'`).

See *The build configuration file* for documentation of all available config values.

2.7 Autodoc

When documenting Python code, it is common to put a lot of documentation in the source files, in documentation strings. Sphinx supports the inclusion of docstrings from your modules with an *extension* (an extension is a Python module that provides additional features for Sphinx projects) called "autodoc".

In order to use autodoc, you need to activate it in `conf.py` by putting the string `'sphinx.ext.autodoc'` into the list assigned to the `extensions` config value. Then, you have a few additional directives at your disposal.

For example, to document the function `io.open()`, reading its signature and docstring from the source file, you'd write this:

```
.. autofunction:: io.open
```

You can also document whole classes or even modules automatically, using member options for the auto directives, like

```
.. automodule:: io
   :members:
```

autodoc needs to import your modules in order to extract the docstrings. Therefore, you must add the appropriate path to `sys.path` in your `conf.py`.

See `sphinx.ext.autodoc` for the complete description of the features of autodoc.

2.8 More topics to be covered

- Other extensions (math, intersphinx, viewcode, doctest)
- Static files
- Selecting a theme
- Templating
- Using extensions
- Writing extensions

CHAPTER

THREE

INVOCATION OF SPHINX-BUILD

The **sphinx-build** script builds a Sphinx documentation set. It is called like this:

```
$ sphinx-build [options] sourcedir builddir [filenames]
```

where *sourcedir* is the *source directory*, and *builddir* is the directory in which you want to place the built documentation. Most of the time, you don't need to specify any *filenames*.

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

-b `buildername`
: The most important option: it selects a builder. The most common builders are:

  html Build HTML pages. This is the default builder.

  dirhtml Build HTML pages, but with a single directory per document. Makes for prettier URLs (no `.html`) if served from a webserver.

  singlehtml Build a single HTML with the whole content.

  htmlhelp, qthelp, devhelp, epub Build HTML files with additional information for building a documentation collection in one of these formats.

  latex Build LaTeX sources that can be compiled to a PDF document using **pdflatex**.

  man Build manual pages in groff format for UNIX systems.

  text Build plain text files.

  doctest Run all doctests in the documentation, if the `doctest` extension is enabled.

  linkcheck Check the integrity of all external links.

  See *Available builders* for a list of all builders shipped with Sphinx. Extensions can add their own builders.

-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.

-t `tag`
: Define the tag *tag*. This is relevant for `only` directives that only include their content if this tag is set. New in version 0.6.

-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.

-C
: Don't look for a configuration file; only take options via the `-D` option. New in version 0.5.

-D setting=value
: Override a configuration value set in the `conf.py` file. The value must be a string or dictionary value. For the latter, supply the setting name and key like this: `-D latex_elements.docclass=scrartcl`. For boolean values, use `0` or `1` as the value. Changed in version 0.6: The value can now be a dictionary value.

-A name=value
: Make the *name* assigned to *value* in the HTML templates. New in version 0.5.

-n
: Run in nit-picky mode. Currently, this generates warnings for all missing references.

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

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

-Q
: Do not output anything on standard output, also suppress warnings. Only errors are written to standard error.

-w file
: Write warnings (and errors) to the given file, in addition to standard error.

-W
: Turn warnings into errors. This means that the build stops at the first warning and `sphinx-build` exits with exit status 1.

-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).

3.1 Makefile options

The `Makefile` and `make.bat` files created by **sphinx-quickstart** usually run **sphinx-build** only with the *-b* and *-d* options. However, they support the following variables to customize behavior:

PAPER
: The value for `latex_paper_size`.

SPHINXBUILD
The command to use instead of `sphinx-build`.

BUILDDIR
The build directory to use instead of the one chosen in **sphinx-quickstart**.

SPHINXOPTS
Additional options for **sphinx-build**.

CHAPTER

FOUR

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. The "ref" links in this document link to the description of the individual constructs in the reST reference.

4.1 Paragraphs

The paragraph (ref) 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.

4.2 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``.

Standard reST provides the following roles:

- emphasis – alternate spelling for `*emphasis*`
- strong – alternate spelling for `**strong**`
- literal – alternate spelling for ``` ``literal`` ```
- subscript – subscript text
- superscript – superscript text
- title-reference – for titles of books, periodicals, and other materials

See *Inline markup* for roles added by Sphinx.

4.3 Lists and Quote-like blocks

List markup (ref) 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.
```

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 (ref) 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.
```

Note that the term cannot have more than one line of text.

Quoted paragraphs (ref) are created by just indenting them more than the surrounding paragraphs.

Line blocks (ref) are a way of preserving line breaks:

```
| These lines are
| broken exactly like in
| the source file.
```

There are also several more special blocks available:

- field lists (ref)
- option lists (ref)
- quoted literal blocks (ref)
- doctest blocks (ref)

4.4 Source Code

Literal code blocks (ref) are introduced by ending a paragraph with the special marker `::`. The literal block must be indented (and, like all paragraphs, separated from the surrounding ones by 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:".

4.5 Tables

Two forms of tables are supported. For *grid tables* (ref), you have to "paint" the cell grid yourself. They look like this:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      |          |
+------------------------+------------+----------+----------+
```

Simple tables (ref) are easier to write, but limited: they must contain more than one row, and the first column cannot contain multiple lines. They look like this:

```
=====  =====  =======
A      B      A and B
=====  =====  =======
False  False  False
True   False  False
False  True   False
```

```
True   True   True
=====  =====  =======
```

4.6 Hyperlinks

4.6.1 External links

Use `Link text <http://example.com/>`_ 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.

You can also separate the link and the target definition (ref), like this:

```
This is a paragraph that contains `a link`_.

.. _a link: http://example.com/
```

4.6.2 Internal links

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

4.7 Sections

Section headers (ref) 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.

4.8 Explicit Markup

"Explicit markup" (ref) 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.)

4.9 Directives

A directive (ref) 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.

Docutils supports the following directives:

- Admonitions: attention, caution, danger, error, hint, important, note, tip, warning and the generic admonition. (Most themes style only "note" and "warning" specially.)
- Images:
  - image (see also Images below)
  - figure (an image with caption and optional legend)
- Additional body elements:
  - contents (a local, i.e. for the current file only, table of contents)
  - container (a container with a custom class, useful to generate an outer `<div>` in HTML)
  - rubric (a heading without relation to the document sectioning)
  - topic, sidebar (special highlighted body elements)
  - parsed-literal (literal block that supports inline markup)
  - epigraph (a block quote with optional attribution line)
  - highlights, pull-quote (block quotes with their own class attribute)
  - compound (a compound paragraph)
- Special tables:
  - table (a table with title)
  - csv-table (a table generated from comma-separated values)
  - list-table (a table generated from a list of lists)
- Special directives:
  - raw (include raw target-format markup)
  - include (include reStructuredText from another file) – in Sphinx, when given an absolute include file path, this directive takes it as relative to the source directory
  - class (assign a class attribute to the next element) [1]

[1] When the default domain contains a `class` directive, this directive will be shadowed. Therefore, Sphinx re-exports it as `rst-class`.

- HTML specifics:
  - meta (generation of HTML `<meta>` tags)
  - title (override document title)
- Influencing markup:
  - default-role (set a new default role)
  - role (create a new role)

  Since these are only per-file, better use Sphinx' facilities for setting the `default_role`.

Do *not* use the directives sectnum, header and footer.

Directives added by Sphinx are described in *Sphinx Markup Constructs*.

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)
   :module: some.module.name

   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 `module` (as you can see, options are given in the lines immediately following the arguments and indicated by the colons). Options must be indented to the same level as the directive content.

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

4.10 Images

reST supports an image directive (ref), used like so:

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

When used within Sphinx, the file name given (here `gnu.png`) must either be relative to the source file, or absolute which means that they are relative to the top source directory. For example, the file `sketch/spam.rst` could refer to the image `images/spam.png` as `../images/spam.png` or `/images/spam.png`.

Sphinx will automatically copy image files over to a subdirectory of the output directory on building (e.g. the `_static` directory for HTML output.)

Interpretation of image size options (`width` and `height`) is as follows: if the size has no unit or the unit is pixels, the given size will only be respected for output channels that support pixels (i.e. not in LaTeX output). Other units (like `pt` for points) will be used for HTML and LaTeX 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.Changed in version 0.6: Image paths can now be absolute.

4.11 Footnotes

For footnotes (ref), use `[#name]_` 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 [#f1]_ dolor sit amet ... [#f2]_

.. rubric:: Footnotes

.. [#f1] Text of the first footnote.
.. [#f2] Text of the second footnote.
```

You can also explicitly number the footnotes (`[1]_`) or use auto-numbered footnotes without names (`[#]_`).

4.12 Citations

Standard reST citations (ref) are supported, with the additional feature that they are "global", i.e. all citations can be referenced from all files. Use them like so:

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

.. [Ref] Book or article reference, URL or whatever.
```

Citation usage is similar to footnote usage, but with a label that is not numeric or begins with `#`.

4.13 Substitutions

reST supports "substitutions" (ref), which are pieces of text and/or markup referred to in the text by `|name|`. They are defined like footnotes with explicit markup blocks, like this:

```
.. |name| replace:: replacement *text*
```

or this:

```
.. |caution| image:: warning.png
              :alt: Warning!
```

See the reST reference for substitutions for details.

If you want to use some substitutions for all documents, put them into `rst_prolog` or put them into a separate file and include it into all documents you want to use them in, using the `include` directive. (Be sure to give the include file a file name extension differing from that of other source files, to avoid Sphinx finding it as a standalone document.)

Sphinx defines some default substitutions, see *Substitutions*.

4.14 Comments

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

```
.. This is a comment.
```

You can indent text after a comment start to form multiline comments:

```
..
   This whole indented block
   is a comment.

   Still in the comment.
```

4.15 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. Sphinx assumes source files to be encoded in UTF-8 by default; you can change this with the `source_encoding` config value.

4.16 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 a backslash-escaped space to get around that. See the reference for the details.
- **No nested inline markup:** Something like `*see :func:`foo`*` is not possible.

CHAPTER

FIVE

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.

5.1 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.

Note: Simple "inclusion" of one file in another can be done with the include directive.

`.. 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. Relative document names (not beginning with a slash) are relative to the document the directive occurs in, absolute names are relative to the source directory. 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.

[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}`.

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 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 also add external links, by giving an HTTP URL instead of a document name.

If you want to have section numbers even in HTML output, give the toctree a `numbered` flag option. For example:

```
.. toctree::
   :numbered:

   foo
   bar
```

Numbering then starts at the heading of `foo`. Sub-toctrees are automatically numbered (don't give the `numbered` flag to those).

If you want only the titles of documents in the tree to show up, not other headings of the same level, you can use the `titlesonly` option:

```
.. toctree::
   :titlesonly:

   foo
   bar
```

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]

The special entry name `self` stands for the document containing the toctree directive. This is useful if you want to generate a "sitemap" from the toctree.

You can also give a "hidden" option to the directive, like this:

```
.. toctree::
   :hidden:
```

[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.

```
doc_1
doc_2
```

This will still notify Sphinx of the document hierarchy, but not insert links into the document at the location of the directive – this makes sense if you intend to insert these links yourself, in a different style, or in the HTML sidebar.

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_docs` to explicitly exclude documents from building, and `exclude_trees` 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.Changed in version 0.6: Added "numbered" and "hidden" options as well as external links and support for "self" references.Changed in version 1.0: Added "titlesonly" option.

5.1.1 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 Python module index, and the search page, respectively.

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

  The Python module index contains one entry per `py: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.)

5.2 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.).

.. **deprecated**:: vesion
Similar to `versionchanged`, but describes when the feature was deprecated. An explanation can also be given, for example to inform the reader what should be used instead. Example:

```
.. deprecated:: 3.1
   Use :func:`spam` instead.
```

.. **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 :py:mod:`zipfile`
      Documentation of the :py:mod:`zipfile` standard module.

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

There's also a "short form" allowed that looks like this:

```
.. seealso:: modules :py:mod:`zipfile`, :py:mod:`tarfile`
```

New in version 0.5: The short form.

.. **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" (or the selected language's equivalent), 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 line of text. Use it as follows:

```
.. centered:: LICENSE AGREEMENT
```

`.. hlist::`
This directive must contain a bullet list. It will transform it into a more compact list by either distributing more than one item horizontally, or reducing spacing between items, depending on the builder.

For builders that support the horizontal distribution, there is a `columns` option that specifies the number of columns; it defaults to 2. Example:

```
.. hlist::
   :columns: 3

   * A list of
   * short items
   * that should be
   * displayed
   * horizontally
```

New in version 0.6.

5.3 Table-of-contents markup

The `toctree` directive, which generates tables of contents of subdocuments, is described in *The TOC tree*.

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

5.4 Index-generating markup

Sphinx automatically creates index entries from all object descriptions (like functions, classes or attributes) like discussed in *Sphinx Domains*.

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`. (These are Python-specific and therefore deprecated.)

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

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

This creates four index entries.

5.5 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.
```

New in version 0.6: You can now give the glossary directive a `:sorted:` flag that will automatically sort the entries alphabetically.

5.6 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:: [name]`
: 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.

  The argument to `productionlist` serves to distinguish different sets of production lists that belong to different grammars.

  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. Outside of the production list, you can reference to token productions using `token`.

  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'
```

5.7 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, but the doc-wide default can be set with the `highlight_language` config value.
- Within Python highlighting mode, interactive sessions are recognized automatically and highlighted appropriately. Normal Python code is only highlighted if it is parseable (so you can use Python as the default, but interspersed snippets of shell commands or other code blocks will not be highlighted as Python).
- 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 when `highlight_language` isn't set)
  - `guess` (let Pygments guess the lexer based on contents, only works with certain well-recognizable languages)
  - `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.

5.7.1 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.
```

5.7.2 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. [3] For example, to include the Python source file `example.py`, use:

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

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

The file name is usually relative to the current file's path. However, if it is absolute (starting with `/`), it is relative to the top source directory.

Tabs in the input are expanded if you give a `tab-width` option with the desired tab width.

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:
```

Include files are assumed to be encoded in the `source_encoding`. If the file has a different encoding, you can specify it with the `encoding` option:

```
.. literalinclude:: example.py
   :encoding: latin-1
```

The directive also supports including only parts of the file. If it is a Python module, you can select a class, function or method to include using the `pyobject` option:

```
.. literalinclude:: example.py
   :pyobject: Timer.start
```

This would only include the code lines belonging to the `start()` method in the `Timer` class within the file.

Alternately, you can specify exactly which lines to include by giving a `lines` option:

```
.. literalinclude:: example.py
   :lines: 1,3,5-10,20-
```

This includes the lines 1, 3, 5 to 10 and lines 20 to the last line.

Another way to control which part of the file is included is to use the `start-after` and `end-before` options (or only one of them). If `start-after` is given as a string option, only lines that follow the first line containing that string are included. If `end-before` is given as a string option, only lines that precede the first lines containing that string are included.

You can prepend and/or append a line to the included code, using the `prepend` and `append` option, respectively. This is useful e.g. for highlighting PHP code that doesn't include the `<?php/?>` markers. New in version 0.4.3: The `encoding` option.New in version 0.6: The `pyobject`, `lines`, `start-after` and `end-before` options, as well as support for absolute filenames.New in version 1.0: The `prepend` and `append` options, as well as `tab-width`.

5.8 Inline markup

Sphinx uses interpreted text roles to insert semantic markup into documents. They are written as `:rolename:`content``.

Note: The default role (`content`) has no special meaning by default. You are free to use it for anything you like, e.g. variable names; use the `default_role` config value to set it to a known role.

See *Sphinx Domains* for roles added by domains.

5.8.1 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.
- If you prefix the content with `~`, the link text will only be the last component of the target. For example, `:py: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.

Cross-referencing objects

These roles are described with their respective domains:

- *Python*
- *C*
- *C++*
- *JavaScript*
- *ReST*

Cross-referencing arbitrary locations

`:ref:`
To support cross-referencing to arbitrary locations in any document, the standard reST labels are used. 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". This works just as well when section and reference are in different source files.

Automatic labels also work with figures: given

```
.. _my-figure:

.. figure:: whatever

   Figure caption
```

a reference `:ref:'my-figure'` would insert a reference to the figure with link text "Figure caption".

The same works for tables that are given an explicit caption using the table directive.

- 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>'`.

Using `ref` is advised over standard reStructuredText links to sections (like `'Section title'_`) because it works across files, when section headings are changed, and for all builders that support cross-references.

Cross-referencing documents

New in version 0.6. There is also a way to directly link to documents:

:doc:
Link to the specified document; the document name can be specified in absolute or relative fashion. For example, if the reference `:doc:'parrot'` occurs in the document `sketches/index`, then the link refers to `sketches/parrot`. If the reference is `:doc:'/people'` or `:doc:'../people'`, the link refers to `people`.

If no explicit link text is given (like usual: `:doc:'Monty Python members </people>'`), the link caption will be the title of the given document.

Referencing downloadable files

New in version 0.6.

:download:
This role lets you link to files within your source tree that are not reST documents that can be viewed, but files that can be downloaded.

When you use this role, the referenced file is automatically marked for inclusion in the output when building (obviously, for HTML output only). All downloadable files are put into the `_downloads` subdirectory of the output directory; duplicate filenames are handled.

An example:

```
See :download:'this example script <../example.py>'.
```

The given filename is usually relative to the directory the current source file is contained in, but if it absolute (starting with `/`), it is taken as relative to the top source directory.

The `example.py` file will be copied to the output directory, and a suitable link generated to it.

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 `option` 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.

5.8.2 Other semantic markup

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

:abbr:
An abbreviation. If the role content contains a parenthesized explanation, it will be treated specially: it will be shown in a tool-tip in HTML, and output only once in LaTeX.

Example: `:abbr:'LIFO (last-in, first-out)'`. New in version 0.6.

: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. Changed in version 1.0: An accelerator key for the GUI label can be included using an ampersand; this will be stripped and displayed underlined in the output (example: `:guilabel:'&Cancel'`). To include a literal ampersand, double it.

: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.

`menuselection` also supports ampersand accelerators just like `guilabel`.

: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`. For example, in `:samp:`print 1+{variable}``, the part `variable` would be emphasized.

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. You can link to a specific section by saying `:pep:`number#anchor``.

: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. You can link to a specific section by saying `:rfc:`number#anchor``.

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

5.8.3 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 (the date on which the document is read), or the date set in the build configuration file. Normally has the format `April 14, 2007`. Set by `today_fmt` and `today`.

5.9 Miscellaneous markup

5.9.1 File-wide metadata

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

```
:fieldname: Field content
```

A field list near the top of a file is parsed by docutils as the "docinfo" which is normally used to record the author, date of publication and other metadata. *In Sphinx*, a field list preceding any other markup is moved from the docinfo to the Sphinx environment as document metadata and is not displayed in the output; a field list appearing after the document title will be part of the docinfo as normal and will be 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.

orphan If set, warnings about this file not being included in any toctree will be suppressed. New in version 1.0.

5.9.2 Meta-information markup

`.. sectionauthor:: name <email>`
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.

`.. codeauthor:: name <email>`
The `codeauthor` directive, which can appear multiple times, names the authors of the described 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.

5.9.3 Including content based on tags

`.. only:: <expression>`
Include the content of the directive only if the *expression* is true. The expression should consist of tags, like this:

```
.. only:: html and draft
```

Undefined tags are false, defined tags (via the `-t` command-line option or within `conf.py`) are true. Boolean expressions, also using parentheses (like `html and (latex or draft)`) are supported.

The format of the current builder (`html`, `latex` or `text`) is always set as a tag. New in version 0.6.

5.9.4 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 list-like elements such as object descriptions, blockquotes or any kind of lists cannot be set out of the box with `tabulary`. They are therefore set with the standard LaTeX `tabular` environment if you don't give a `tabularcolumns` directive. If you do, the table will be set with `tabulary`, but you must use the `p{width}` construct for the columns that contain these elements. Literal blocks do not work with `tabulary` at all, so tables containing a literal block are always set with `tabular`. 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.

More markup is added by *Sphinx Domains*.

CHAPTER

SIX

SPHINX DOMAINS

New in version 1.0.

6.1 What is a Domain?

Originally, Sphinx was conceived for a single project, the documentation of the Python language. Shortly afterwards, it was made available for everyone as a documentation tool, but the documentation of Python modules remained deeply built in – the most fundamental directives, like `function`, were designed for Python objects. Since Sphinx has become somewhat popular, interest developed in using it for many different purposes: C/C++ projects, JavaScript, or even reStructuredText markup (like in this documentation).

While this was always possible, it is now much easier to easily support documentation of projects using different programming languages or even ones not supported by the main Sphinx distribution, by providing a **domain** for every such purpose.

A domain is a collection of markup (reStructuredText *directive*s and *role*s) to describe and link to *object*s belonging together, e.g. elements of a programming language. Directive and role names in a domain have names like `domain:name`, e.g. `py:function`. Domains can also provide custom indices (like the Python Module Index).

Having domains means that there are no naming problems when one set of documentation wants to refer to e.g. C++ and Python classes. It also means that extensions that support the documentation of whole new languages are much easier to write.

This section describes what the domains that come with Sphinx provide. The domain API is documented as well, in the section *Domain API*.

6.2 Basic Markup

Most domains provide a number of *object description directives*, used to describe specific objects 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:`. An example using a Python domain directive:

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

   Spam or ham the foo.
```

This describes the two Python functions `spam` and `ham`. (Note that when signatures become too long, you can break them if you add a backslash to lines that are continued in the next line. Example:

```
.. py:function:: filterwarnings(action, message='', category=Warning, \
                                module='', lineno=0, append=False)
   :noindex:
```

(This example also shows how to use the `:noindex:` flag.)

The domains also provide roles that link back to these object descriptions. For example, to link to one of the functions described in the example above, you could say

```
The function :py:func:`spam` does a similar thing.
```

As you can see, both directive and role names contain the domain name and the directive name.

Default Domain

To avoid having to writing the domain name all the time when you e.g. only describe Python objects, a default domain can be selected with either the config value `primary_domain` or this directive:

`.. default-domain:: name`
: Select a new default domain. While the `primary_domain` selects a global default, this only has an effect within the same file.

If no other default is selected, the Python domain (named `py`) is the default one, mostly for compatibility with documentation written for older versions of Sphinx.

Directives and roles that belong to the default domain can be mentioned without giving the domain name, i.e.

```
.. function:: pyfunc()

   Describes a Python function.

Reference to :func:`pyfunc`.
```

6.2.1 Cross-referencing syntax

For cross-reference roles provided by domains, the same facilities exist as for general cross-references. See *Cross-referencing syntax*.

In short:

- You may supply an explicit title and reference target: `: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.
- If you prefix the content with `~`, the link text will only be the last component of the target. For example, `:py:meth:`~Queue.Queue.get`` will refer to `Queue.Queue.get` but only display `get` as the link text.

6.3 The Python Domain

The Python domain (name **py**) provides the following directives for module declarations:

.. **py: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. py: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.

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

The following directives are provided for module and class contents:

.. **py: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.

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

.. **py: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 *Python Signatures*. For example:

```
.. py: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.

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

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

```
-- or --

.. py:class:: Bar

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

The first way is the preferred one.

.. **py: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.

.. **py: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 *Python Signatures*.

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

.. **py:classmethod::** name(signature)
Like `py:method`, but indicates that the method is a class method. New in version 0.6.

6.3.1 Python 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:

```
.. py: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:

```
.. py: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:

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

6.3.2 Info field lists

New in version 0.4. Inside Python object description 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:

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

   Format the exception with a traceback.

   :param etype: 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

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

Return type list of strings

It is also possible to combine parameter type and description, if the type is a single word, like this:

```
:param integer limit: maximum number of stack frames to show
```

6.3.3 Cross-referencing Python objects

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

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

:py:func:
Reference 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).

:py:data:
Reference a module-level variable.

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

:py:class:
Reference a class; a dotted name may be used.

:py:meth:
Reference a method of an object. The role text can 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.

:py:attr:
Reference a data attribute of an object.

:py:exc:
Reference an exception. A dotted name may be used.

:py:obj:
Reference 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, `:py:func:`filter`` could refer to a function named `filter` in the current module, or the built-in function of that name. In contrast, `:py: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, `:py:func:`open`` always refers to the built-in function, while `:py:func:`.open`` refers to `codecs.open()`.

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

Also, if the name is prefixed with a dot, and no exact match is found, the target is taken as a suffix and all object names with that suffix are searched. For example, `:py:meth:`.TarFile.close`` references the `tarfile.TarFile.close()` function, even if the current module is not `tarfile`. Since this can get ambiguous, if there is more than one possible match, you will get a warning from Sphinx.

Note that you can combine the `~` and `.` prefixes: `:py:meth:`~.TarFile.close`` will reference the `tarfile.TarFile.close()` method, but the visible link caption will only be `close()`.

6.4 The C Domain

The C domain (name **c**) is suited for documentation of C API.

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

```
.. c:function:: 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.

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

```
.. c:member:: 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.

.. c:macro:: `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`.

.. **c:type::** name
Describes a C type (whether defined by a typedef or struct). The signature should just be the type name.

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

```
.. c:var:: PyObject* PyClass_Type
```

6.4.1 Cross-referencing C constructs

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

:c:data:
Reference a C-language variable.

:c:func:
Reference a C-language function. Should include trailing parentheses.

:c:macro:
Reference a "simple" C macro, as defined above.

:c:type:
Reference a C-language type.

6.5 The C++ Domain

The C++ domain (name **cpp**) supports documenting C++ projects.

The following directives are available:

.. **cpp:class::** signatures
.. **cpp:function::** signatures
.. **cpp:member::** signatures
.. **cpp:type::** signatures
Describe a C++ object. Full signature specification is supported – give the signature as you would in the declaration. Here some examples:

```
.. cpp:function:: bool namespaced::theclass::method(int arg1, std::string arg2)

   Describes a method with parameters and types.

.. cpp:function:: bool namespaced::theclass::method(arg1, arg2)

   Describes a method without types.

.. cpp:function:: const T &array<T>::operator[]() const

   Describes the constant indexing operator of a templated array.

.. cpp:function:: operator bool() const

   Describe a casting operator here.

.. cpp:function:: constexpr void foo(std::string &bar[2]) noexcept
```

```
   Describe a constexpr function here.

.. cpp:member:: std::string theclass::name

.. cpp:member:: std::string theclass::name[N][M]

.. cpp:type:: theclass::const_iterator
```

Will be rendered like this:

bool namespaced::theclass::**method**(int *arg1*, std::string *arg2*)
Describes a method with parameters and types.

bool namespaced::theclass::**method**(*arg1*, *arg2*)
Describes a method without types.

const T& array<T>::**operator[]**() const
Describes the constant indexing operator of a templated array.

operator bool() const
Describe a casting operator here.

constexpr void **foo**(std::string& *bar*[2]) noexcept
Describe a constexpr function here.

std::string theclass::**name**

std::string theclass::**name**[N][M]

type theclass::**const_iterator**

.. **cpp:namespace::** namespace
Select the current C++ namespace for the following objects.

These roles link to the given object types:

:cpp:class:
:cpp:func:
:cpp:member:
:cpp:type:
Reference a C++ object. You can give the full signature (and need to, for overloaded functions.)

Note: Sphinx' syntax to give references a custom title can interfere with linking to template classes, if nothing follows the closing angle bracket, i.e. if the link looks like this: `:cpp:class:`MyClass<T>``. This is interpreted as a link to `T` with a title of `MyClass`. In this case, please escape the opening angle bracket with a backslash, like this: `:cpp:class:`MyClass\<T>``.

Note on References

It is currently impossible to link to a specific version of an overloaded method. Currently the C++ domain is the first domain that has basic support for overloaded methods and until there is more data for comparison we don't want to select a bad syntax to reference a specific overload. Currently Sphinx will link to the first overloaded version of the method / function.

6.6 The Standard Domain

The so-called "standard" domain collects all markup that doesn't warrant a domain of its own. Its directives and roles are not prefixed with a domain name.

The standard domain is also where custom object descriptions, added using the `add_object_type()` API, are placed.

There is a set of directives allowing documenting command-line programs:

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

```
.. option:: -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 or program uses or defines. Referencable by `envvar`.

`.. program:: name`
Like `py:currentmodule`, this directive produces no output. Instead, it serves to notify Sphinx that all following `option` directives document options for the program called *name*.

If you use `program`, you have to qualify the references in your `option` roles by the program name, so if you have the following situation

```
.. program:: rm

.. option:: -r

   Work recursively.

.. program:: svn

.. option:: -r revision

   Specify the revision to work upon.
```

then `:option:`rm -r`` would refer to the first option, while `:option:`svn -r`` would refer to the second one.

The program name may contain spaces (in case you want to document subcommands like `svn add` and `svn commit` separately). New in version 0.5.

There is also a very generic object description directive, which is not tied to any domain:

`.. describe:: text`
`.. object:: text`
This directive produces the same formatting as the specific ones provided by domains, but does not create index entries or cross-referencing targets. Example:

```
.. describe:: PAPER

   You can set this variable to select a paper size.
```

6.7 The JavaScript Domain

The JavaScript domain (name **js**) provides the following directives:

.. **js:function::** name(signature)
Describes a JavaScript function or method. If you want to describe arguments as optional use square brackets as *documented* for Python signatures.

You can use fields to give more details about arguments and their expected types, errors which may be thrown by the function, and the value being returned:

```
.. js:function:: $.getJSON(href, callback[, errback])

   :param string href: An URI to the location of the resource.
   :param callback: Get's called with the object.
   :param errback:
       Get's called in case the request fails. And a lot of other
       text so we need multiple lines
   :throws SomeError: For whatever reason in that case.
   :returns: Something
```

This is rendered as:

$.**getJSON**(*href*, *callback*[, *errback*])
Arguments
- **href** (*string*) – An URI to the location of the resource.
- **callback** – Get's called with the object.
- **errback** – Get's called in case the request fails. And a lot of other text so we need multiple lines.

Throws SomeError For whatever reason in that case.
Returns Something

.. **js:class::** name
Describes a constructor that creates an object. This is basically like a function but will show up with a *class* prefix:

```
.. js:class:: MyAnimal(name[, age])

   :param string name: The name of the animal
   :param number age: an optional age for the animal
```

This is rendered as:

class **MyAnimal**(*name*[, *age*])
Arguments
- **name** (*string*) – The name of the animal
- **age** (*number*) – an optional age for the animal

.. **js:data::** name
Describes a global variable or constant.

.. **js:attribute::** object.name
Describes the attribute *name* of *object*.

These roles are provided to refer to the described objects:

:js:func:
:js:class:
:js:data:
:js:attr:

6.8 The reStructuredText domain

The reStructuredText domain (name **rst**) provides the following directives:

.. rst:directive:: name
Describes a reST directive. The *name* can be a single directive name or actual directive syntax (.. prefix and :: suffix) with arguments that will be rendered differently. For example:

```
.. rst:directive:: foo

   Foo description.

.. rst:directive:: .. bar:: baz

   Bar description.
```

will be rendered as:

> **.. foo::**
> Foo description.
>
> **.. bar::** baz
> Bar description.

.. rst:role:: name
Describes a reST role. For example:

```
.. rst:role:: foo

   Foo description.
```

will be rendered as:

> **:foo:**
> Foo description.

These roles are provided to refer to the described objects:

:rst:dir:
:rst:role:

6.9 More domains

The sphinx-contrib repository contains more domains available as extensions; currently a Ruby and an Erlang domain.

CHAPTER

SEVEN

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 sphinx.builders.html.**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 sphinx.builders.html.**DirectoryHTMLBuilder**
This is a subclass of the standard HTML builder. Its output is a directory with HTML files, where each file is called `index.html` and placed in a subdirectory named like its page name. For example, the document `markup/rest.rst` will not result in an output file `markup/rest.html`, but `markup/rest/index.html`. When generating links between pages, the `index.html` is omitted, so that the URL would look like `markup/rest/`.

Its name is `dirhtml`. New in version 0.6.

class sphinx.builders.html.**SingleFileHTMLBuilder**
This is an HTML builder that combines the whole project in one output file. (Obviously this only works with smaller projects.) The file is named like the master document. No indices will be generated.

Its name is `singlehtml`. New in version 1.0.

class sphinx.builders.htmlhelp.**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 sphinx.builders.qthelp.**QtHelpBuilder**
This builder produces the same output as the standalone HTML builder, but also generates Qt help collection support files that allow the Qt collection generator to compile them.

Its name is `qthelp`.

class sphinx.builders.devhelp.**DevhelpBuilder**
This builder produces the same output as the standalone HTML builder, but also generates GNOME Devhelp support file that allows the GNOME Devhelp reader to view them.

Its name is `devhelp`.

class sphinx.builders.epub.**EpubBuilder**
This builder produces the same output as the standalone HTML builder, but also generates an *epub*

file for ebook readers. See *Epub info* for details about it. For definition of the epub format, have a look at http://www.idpf.org/specs.htm or http://en.wikipedia.org/wiki/EPUB.

Some ebook readers do not show the link targets of references. Therefore this builder adds the targets after the link when necessary. The display of the URLs can be customized by adding CSS rules for the class `link-target`.

Its name is `epub`.

class `sphinx.builders.latex.`**`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.

Note: The produced LaTeX file uses several LaTeX packages that may not be present in a "minimal" TeX distribution installation. For TeXLive, the following packages need to be installed:

- latex-recommended
- latex-extra
- fonts-recommended

Its name is `latex`.

Note that a direct PDF builder using ReportLab is available in rst2pdf version 0.12 or greater. You need to add `'rst2pdf.pdfbuilder'` to your `extensions` to enable it, its name is `pdf`. Refer to the rst2pdf manual for details.

class `sphinx.builders.text.`**`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 `sphinx.builders.manpage.`**`ManualPageBuilder`**
This builder produces manual pages in the groff format. You have to specify which documents are to be included in which manual pages via the `man_pages` configuration value.

Its name is `man`.

Note: This builder requires the docutils manual page writer, which is only available as of docutils 0.6.

New in version 1.0.

class `sphinx.builders.html.`**`SerializingHTMLBuilder`**
This builder uses a module that implements the Python serialization API (*pickle*, *simplejson*, *phpserialize*, and others) to dump the generated HTML documentation. The pickle builder is a subclass of it.

A concrete subclass of this builder serializing to the PHP serialization format could look like this:

```
import phpserialize

class PHPSerializedBuilder(SerializingHTMLBuilder):
    name = 'phpserialized'
    implementation = phpserialize
    out_suffix = '.file.phpdump'
```

```
globalcontext_filename = 'globalcontext.phpdump'
searchindex_filename = 'searchindex.phpdump'
```

implementation
A module that implements *dump()*, *load()*, *dumps()* and *loads()* functions that conform to the functions with the same names from the pickle module. Known modules implementing this interface are *simplejson* (or *json* in Python 2.6), *phpserialize*, *plistlib*, and others.

out_suffix
The suffix for all regular files.

globalcontext_filename
The filename for the file that contains the "global context". This is a dict with some general configuration values such as the name of the project.

searchindex_filename
The filename for the search index Sphinx generates.

See *Serialization builder details* for details about the output format. New in version 0.5.

class sphinx.builders.html.**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.

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

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

The file suffix is `.fpickle`. The global context is called `globalcontext.pickle`, the search index `searchindex.pickle`.

class sphinx.builders.html.**JSONHTMLBuilder**
This builder produces a directory with JSON 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.

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

Its name is `json`.

The file suffix is `.fjson`. The global context is called `globalcontext.json`, the search index `searchindex.json`. New in version 0.5.

class sphinx.builders.changes.**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 sphinx.builders.linkcheck.**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`

7.1 Serialization builder details

All serialization builders outputs one file per source file and a few special files. They also copy the reST source files in the directory `_sources` under the output directory.

The `PickleHTMLBuilder` is a builtin subclass that implements the pickle serialization interface.

The files per source file have the extensions of `out_suffix`, and are arranged in directories just as the source files are. They unserialize to a dictionary (or dictionary like structure) 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:

SerializingHTMLBuilder.globalcontext_filename A pickled dict with these keys:

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

style `html_style`.

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.

SerializingHTMLBuilder.searchindex_filename 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.

environment.pickle The build environment. This is always a pickle file, independent of the builder and a copy of the environment that was used when the builder was started.

Todo

Document common members.

Unlike the other pickle files this pickle file requires that the `sphinx` package is available on unpickling.

CHAPTER

EIGHT

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 is 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.builders.Builder"` means the `Builder` class in the `sphinx.builders` 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.
- There is a special object named `tags` available in the config file. It can be used to query and change the tags (see *Including content based on tags*). Use `tags.has('tag')` to query, `tags.add('tag')` and `tags.remove('tag')` to change.

8.1 General configuration

`extensions`
A list of strings that are module names of Sphinx extensions. These can be extensions coming with Sphinx (named `sphinx.ext.*`) 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'`.

source_encoding
The encoding of all reST source files. The recommended encoding, and the default value, is `'utf-8-sig'`. New in version 0.5: Previously, Sphinx accepted only UTF-8 encoded sources.

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

exclude_patterns
A list of glob-style patterns that should be excluded when looking for source files. [1] They are matched against the source file names relative to the source directory, using slashes as directory separators on all platforms.

Example patterns:

- `'library/xml.rst'` – ignores the `library/xml.rst` file (replaces entry in `unused_docs`)
- `'library/xml'` – ignores the `library/xml` directory (replaces entry in `exclude_trees`)
- `'library/xml*'` – ignores all files and directories starting with `library/xml`
- `'**/.svn'` – ignores all `.svn` directories (replaces entry in `exclude_dirnames`)

`exclude_patterns` is also consulted when looking for static files in `html_static_path`. New in version 1.0.

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. Deprecated since version 1.0: Use `exclude_patterns` instead.

exclude_trees
A list of directory paths, relative to the source directory, that are to be recursively excluded from the search for source files, that is, their subdirectories won't be searched too. The default is `[]`. New in version 0.4.Deprecated since version 1.0: Use `exclude_patterns` instead.

exclude_dirnames
A list of directory names that are to be excluded from any recursive operation Sphinx performs (e.g. searching for source files or copying static files). This is useful, for example, to exclude version-control-specific directories like `'CVS'`. The default is `[]`. New in version 0.5.Deprecated since version 1.0: Use `exclude_patterns` instead.

locale_dirs
New in version 0.5. Directories in which to search for additional Sphinx message catalogs (see

[1] 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.

language), relative to the source directory. The directories on this path are searched by the standard gettext module for a text domain of sphinx; so if you add the directory ./locale to this settting, the message catalogs (compiled from .po format using **msgfmt**) must be in ./locale/language/LC_MESSAGES/sphinx.mo.

The default is [].

templates_path
A list of paths that contain extra templates (or templates that overwrite builtin/theme-specific 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). (Note that the template bridge must be made theme-aware if HTML themes are to be used.)

rst_epilog
A string of reStructuredText that will be included at the end of every source file that is read. This is the right place to add substitutions that should be available in every file. An example:

```
rst_epilog = """
.. |psf| replace:: Python Software Foundation
"""
```

New in version 0.6.

rst_prolog
A string of reStructuredText that will be included at the beginning of every source file that is read. New in version 1.0.

primary_domain
The name of the default *domain*. Can also be None to disable a default domain. The default is 'py'. Those objects in other domains (whether the domain name is given explicitly, or selected by a default-domain directive) will have the domain name explicitly prepended when named (e.g., when the default domain is C, Python functions will be named "Python function", not just "function"). New in version 1.0.

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 'py:obj' to make `filter` a cross-reference to the Python 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.

keep_warnings
If true, keep warnings as "system message" paragraphs in the built documents. Regardless of this setting, warnings are always written to the standard error stream when sphinx-build is run.

The default is False, the pre-0.5 behavior was to always keep them. New in version 0.5.

needs_sphinx
If set to a major.minor version string like '1.1', Sphinx will compare it with its version and refuse to build if it is too old. Default is no requirement. New in version 1.0.

nitpicky
If true, Sphinx will warn about *all* references where the target cannot be found. Default is False. You can activate this mode temporarily using the *-n* command-line switch. New in version 1.0.

8.2 Project information

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.

language
The code for the language the docs are written in. Any text automatically generated by Sphinx will be in that language. Also, in the LaTeX builder, a suitable language will be selected as an option for the *Babel* package. Default is `None`, which means that no translation will be done. New in version 0.5. Currently supported languages are:

- `bn` – Bengali
- `ca` – Catalan
- `cs` – Czech
- `da` – Danish
- `de` – German
- `en` – English
- `es` – Spanish
- `fi` – Finnish
- `fr` – French
- `hr` – Croatian
- `it` – Italian
- `ja` – Japanese
- `lt` – Lithuanian
- `nl` – Dutch
- `pl` – Polish
- `pt_BR` – Brazilian Portuguese
- `ru` – Russian
- `sl` – Slovenian
- `tr` – Turkish
- `uk_UA` – Ukrainian
- `zh_CN` – Simplified Chinese

- zh_TW – Traditional Chinese

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' (or, if translation is enabled with language, am equivalent %format for the selected locale).

highlight_language
The default language to highlight source code in. The default is 'python'. The value should be a valid Pygments lexer name, see *Showing code examples* for more details. New in version 0.5.

pygments_style
The style name to use for Pygments highlighting of source code. The default style is selected by the theme for HTML output, and 'sphinx' otherwise. 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.

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 *object* names (for object types where a "module" of some kind is defined), e.g. for py:function directives. Default is True.

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

modindex_common_prefix
A list of prefixes that are ignored for sorting the Python module index (e.g., if this is set to ['foo.'], then foo.bar is shown under B, not F). This can be handy if you document a project that consists of a single package. Works only for the HTML builder currently. Default is []. New in version 0.6.

trim_footnote_reference_space
Trim spaces before footnote references that are necessary for the reST parser to recognize the footnote, but do not look too nice in the output. New in version 0.6.

trim_doctest_flags
If true, doctest flags (comments looking like # doctest: FLAG, ...) at the ends of lines are removed for all code blocks showing interactive Python sessions (i.e. doctests). Default is true. See the extension doctest for more possibilities of including doctests. New in version 1.0.

8.3 Options for HTML output

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

html_theme
The "theme" that the HTML output should use. See the *section about theming*. The default is 'default'. New in version 0.6.

html_theme_options
A dictionary of options that influence the look and feel of the selected theme. These are theme-specific. For the options understood by the builtin themes, see *this section*. New in version 0.6.

html_theme_path
A list of paths that contain custom themes, either as subdirectories or as zip files. Relative paths are taken as relative to the configuration directory. New in version 0.6.

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 the stylesheet given by the selected theme. If you only want to add or override a few things compared to the theme's stylesheet, use CSS `@import` to import the theme's stylesheet.

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_context
A dictionary of values to pass into the template engine's context for all pages. Single values can also be put in this dictionary using the `-A` command-line option of `sphinx-build`. New in version 0.5.

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 theme's static files, so a file named `default.css` will overwrite the theme's `default.css`. Changed in version 0.4: The paths in `html_static_path` can now contain subdirectories.Changed in version 1.0: The entries in `html_static_path` can now be single files.

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'` (or a locale-dependent equivalent).

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

html_add_permalinks
If true, Sphinx will add "permalinks" for each heading and description environment as paragraph signs that become visible when the mouse hovers over them. Default: `True`. New in version 0.6: Previously, this was always activated.

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

The keys can contain glob-style patterns [1], in which case all matching documents will get the specified sidebars. (A warning is emitted when a more than one glob-style pattern matches for any document.)

The values can be either lists or single strings.

- If a value is a list, it specifies the complete list of sidebar templates to include. If all or some of the default sidebars are to be included, they must be put into this list as well.

  The default sidebars (for documents that don't match any pattern) are: `['localtoc.html', 'relations.html', 'sourcelink.html', 'searchbox.html']`.

- If a value is a single string, it specifies a custom sidebar to be added between the `'sourcelink.html'` and `'searchbox.html'` entries. This is for compatibility with Sphinx versions before 1.0.

Builtin sidebar templates that can be rendered are:

- **localtoc.html** – a fine-grained table of contents of the current document
- **globaltoc.html** – a coarse-grained table of contents for the whole documentation set, collapsed
- **relations.html** – two links to the previous and next documents
- **sourcelink.html** – a link to the source of the current document, if enabled in `html_show_sourcelink`
- **searchbox.html** – the "quick search" box

Example:

```
html_sidebars = {
   '**': ['globaltoc.html', 'sourcelink.html', 'searchbox.html'],
   'using/windows': ['windowssidebar.html', 'searchbox.html'],
}
```

This will render the custom template `windowssidebar.html` and the quick search box within the sidebar of the given document, and render the default sidebars for all other pages (except that the local TOC is replaced by the global TOC). New in version 1.0: The ability to use globbing keys and to specify multiple sidebars. Note that this value only has no effect if the chosen theme does not possess a sidebar, like the builtin **scrolls** and **haiku** themes.

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`.

html_domain_indices
If true, generate domain-specific indices in addition to the general index. For e.g. the Python domain, this is the global module index. Default is `True`.

This value can be a bool or a list of index names that should be generated. To find out the index name for a specific index, look at the HTML file name. For example, the Python module index has the name `'py-modindex'`. New in version 1.0.

html_use_modindex
If true, add a module index to the HTML documents. Default is `True`. Deprecated since version 1.0: Use `html_domain_indices`.

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`.

> **Warning:** If this config value is set to `False`, the JavaScript search function will only display the titles of matching documents, and no excerpt from the matching contents.

html_show_sourcelink
If true (and `html_copy_source` is true as well), links to the reST sources will be added to the sidebar. The default is `True`. New in version 0.6.

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
This is the file name suffix for generated HTML files. The default is `".html"`. New in version 0.4.

html_link_suffix
Suffix for generated links to HTML files. The default is whatever `html_file_suffix` is set to; it can be set differently (e.g. to support different web server setups). New in version 0.6.

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_copyright
If true, "(C) Copyright ..." is shown in the HTML footer. Default is `True`. New in version 1.0.

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

html_output_encoding
Encoding of HTML output files. Default is `'utf-8'`. Note that this encoding name must both be a valid Python encoding name and a valid HTML `charset` value. New in version 1.0.

html_compact_lists
If true, list items containing only a single paragraph will not be rendered with a `<p>` element. This is standard docutils behavior. Default: `True`. New in version 1.0.

html_secnumber_suffix
Suffix for section numbers. Default: `". "`. Set to `" "` to suppress the final dot on section numbers. New in version 1.0.

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

8.4 Options for epub output

These options influence the epub output. As this builder derives from the HTML builder, the HTML options also apply where appropriate. The actual values for some of the options is not really important, they just have to be entered into the Dublin Core metadata.

epub_basename
The basename for the epub file. It defaults to the `project` name.

epub_theme
The HTML theme for the epub output. Since the default themes are not optimized for small screen space, using the same theme for HTML and epub output is usually not wise. This defaults to `'epub'`, a theme designed to save visual space.

epub_title
The title of the document. It defaults to the `html_title` option but can be set independently for epub creation.

epub_author
The author of the document. This is put in the Dublin Core metadata. The default value is `'unknown'`.

epub_language
The language of the document. This is put in the Dublin Core metadata. The default is the `language` option or `'en'` if unset.

epub_publisher
The publisher of the document. This is put in the Dublin Core metadata. You may use any sensible string, e.g. the project homepage. The default value is `'unknown'`.

epub_copyright
The copyright of the document. It defaults to the `copyright` option but can be set independently for epub creation.

epub_identifier
An identifier for the document. This is put in the Dublin Core metadata. For published documents this is the ISBN number, but you can also use an alternative scheme, e.g. the project homepage. The default value is `'unknown'`.

epub_scheme
The publication scheme for the `epub_identifier`. This is put in the Dublin Core metadata. For published books the scheme is `'ISBN'`. If you use the project homepage, `'URL'` seems reasonable. The default value is `'unknown'`.

epub_uid
A unique identifier for the document. This is put in the Dublin Core metadata. You may use a random string. The default value is `'unknown'`.

epub_pre_files
Additional files that should be inserted before the text generated by Sphinx. It is a list of tuples containing the file name and the title. If the title is empty, no entry is added to `toc.ncx`. Example:

```
epub_pre_files = [
    ('index.html', 'Welcome'),
]
```

The default value is `[]`.

epub_post_files
Additional files that should be inserted after the text generated by Sphinx. It is a list of tuples con-

taining the file name and the title. This option can be used to add an appendix. If the title is empty, no entry is added to `toc.ncx`. The default value is `[]`.

epub_exclude_files
A list of files that are generated/copied in the build directory but should not be included in the epub file. The default value is `[]`.

epub_tocdepth
The depth of the table of contents in the file `toc.ncx`. It should be an integer greater than zero. The default value is 3. Note: A deeply nested table of contents may be difficult to navigate.

epub_tocdup
This flag determines if a toc entry is inserted again at the beginning of it's nested toc listing. This allows easier navitation to the top of a chapter, but can be confusing because it mixes entries of differnet depth in one list. The default value is `True`.

8.5 Options for LaTeX output

These options influence LaTeX output.

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*. This is inserted as LaTeX markup, so special characters like a backslash or ampersand must be represented by the proper LaTeX commands if they are to be inserted literally.
- *author*: Author for the LaTeX document. The same LaTeX markup caveat as for *title* applies. Use `\and` to separate multiple authors, as in: `'John \and Sarah'`.
- *documentclass*: Normally, one of `'manual'` or `'howto'` (provided by Sphinx). Other document classes can be given, but they must include the "sphinx" package in order to define Sphinx' custom LaTeX commands. "howto" documents will not 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_domain_indices
If true, generate domain-specific indices in addition to the general index. For e.g. the Python domain, this is the global module index. Default is `True`.

This value can be a bool or a list of index names that should be generated, like for `html_domain_indices`. New in version 1.0.

latex_use_modindex
If true, add a module index to LaTeX documents. Default is `True`. Deprecated since version 1.0: Use `latex_domain_indices`.

latex_show_pagerefs
If true, add page references after internal references. This is very useful for printed copies of the manual. Default is `False`. New in version 1.0.

latex_show_urls
If true, add URL addresses after links. This is very useful for printed copies of the manual. Default is `False`. New in version 1.0.

latex_elements
New in version 0.5. A dictionary that contains LaTeX snippets that override those Sphinx usually puts into the generated `.tex` files.

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

- Keys that you may want to override include:

  'papersize' Paper size option of the document class (`'a4paper'` or `'letterpaper'`), default `'letterpaper'`.

  'pointsize' Point size option of the document class (`'10pt'`, `'11pt'` or `'12pt'`), default `'10pt'`.

  'babel' "babel" package inclusion, default `'\\usepackage{babel}'`.

  'fontpkg' Font package inclusion, default `'\\usepackage{times}'` (which uses Times and Helvetica). You can set this to `''` to use the Computer Modern fonts.

  'fncychap' Inclusion of the "fncychap" package (which makes fancy chapter titles), default `'\\usepackage[Bjarne]{fncychap}'` for English documentation, `'\\usepackage[Sonny]{fncychap}'` for internationalized docs (because the "Bjarne" style uses numbers spelled out in English). Other "fncychap" styles you can try include "Lenny", "Glenn", "Conny" and "Rejne". You can also set this to `''` to disable fncychap.

  'preamble' Additional preamble content, default empty.

  'footer' ` Additional footer content (before the indices), default empty.

- Keys that don't need be overridden unless in special cases are:

  'inputenc' "inputenc" package inclusion, default `'\\usepackage[utf8]{inputenc}'`.

  'fontenc' "fontenc" package inclusion, default `'\\usepackage[T1]{fontenc}'`.

  'maketitle' "maketitle" call, default `'\\maketitle'`. Override if you want to generate a differently-styled title page.

  'tableofcontents' "tableofcontents" call, default `'\\tableofcontents'`. Override if you want to generate a different table of contents or put content between the title page and the TOC.

  'printindex' "printindex" call, the last thing in the file, default `'\\printindex'`. Override if you want to generate the index differently or append some content after the index.

- Keys that are set by other options and therefore should not be overridden are:

  `'docclass' 'classoptions' 'title' 'date' 'release' 'author' 'logo' 'releasename' 'makeindex' 'shorthandoff'`

latex_docclass
A dictionary mapping `'howto'` and `'manual'` to names of real document classes that will be used as the base for the two Sphinx classes. Default is to use `'article'` for `'howto'` and `'report'` for `'manual'`. New in version 1.0.

latex_additional_files
A list of file names, relative to the configuration directory, to copy to the build directory when building LaTeX output. This is useful to copy files that Sphinx doesn't copy automatically, e.g. if they are referenced in custom LaTeX added in `latex_elements`. Image files that are referenced in source files (e.g. via `.. image::`) are copied automatically.

You have to make sure yourself that the filenames don't collide with those of any automatically copied files. New in version 0.6.

latex_preamble
Additional LaTeX markup for the preamble. Deprecated since version 0.5: Use the `'preamble'` key in the `latex_elements` value.

latex_paper_size
The output paper size (`'letter'` or `'a4'`). Default is `'letter'`. Deprecated since version 0.5: Use the `'papersize'` key in the `latex_elements` value.

latex_font_size
The font size ('10pt', '11pt' or '12pt'). Default is `'10pt'`. Deprecated since version 0.5: Use the `'pointsize'` key in the `latex_elements` value.

8.6 Options for manual page output

These options influence manual page output.

man_pages
This value determines how to group the document tree into manual pages. It must be a list of tuples `(startdocname, name, description, authors, section)`, where the items are:

- *startdocname*: document name that is the "root" of the manual page. All documents referenced by it in TOC trees will be included in the manual file too. (If you want one master manual page, use your `master_doc` here.)
- *name*: name of the manual page. This should be a short string without spaces or special characters. It is used to determine the file name as well as the name of the manual page (in the NAME section).
- *description*: description of the manual page. This is used in the NAME section.
- *authors*: A list of strings with authors, or a single string. Can be an empty string or list if you do not want to automatically generate an AUTHORS section in the manual page.
- *section*: The manual page section. Used for the output file name as well as in the manual page header.

New in version 1.0.

CHAPTER

NINE

HTML THEMING SUPPORT

New in version 0.6. Sphinx supports changing the appearance of its HTML output via *themes*. A theme is a collection of HTML templates, stylesheet(s) and other static files. Additionally, it has a configuration file which specifies from which theme to inherit, which highlighting style to use, and what options exist for customizing the theme's look and feel.

Themes are meant to be project-unaware, so they can be used for different projects without change.

9.1 Using a theme

Using an existing theme is easy. If the theme is builtin to Sphinx, you only need to set the `html_theme` config value. With the `html_theme_options` config value you can set theme-specific options that change the look and feel. For example, you could have the following in your `conf.py`:

```
html_theme = "default"
html_theme_options = {
    "rightsidebar": "true",
    "relbarbgcolor": "black"
}
```

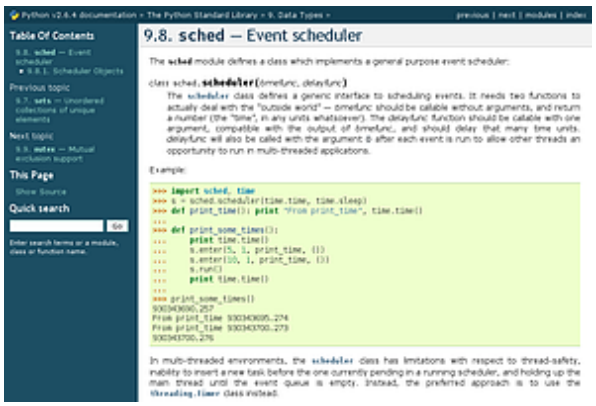
That would give you the default theme, but with a sidebar on the right side and a black background for the relation bar (the bar with the navigation links at the page's top and bottom).

If the theme does not come with Sphinx, it can be in two forms: either a directory (containing `theme.conf` and other needed files), or a zip file with the same contents. Either of them must be put where Sphinx can find it; for this there is the config value `html_theme_path`. It gives a list of directories, relative to the directory containing `conf.py`, that can contain theme directories or zip files. For example, if you have a theme in the file `blue.zip`, you can put it right in the directory containing `conf.py` and use this configuration:

```
html_theme = "blue"
html_theme_path = ["."]
```

9.2 Builtin themes

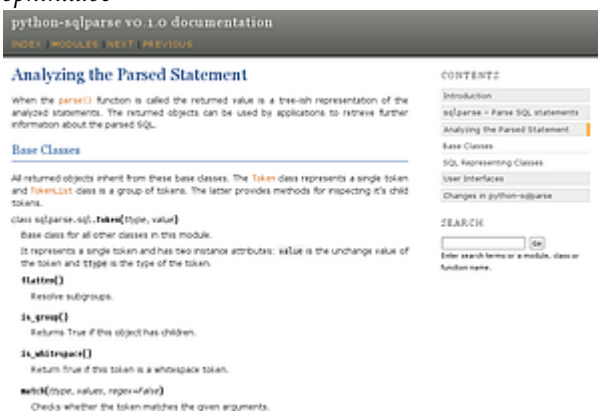
Theme overview



default

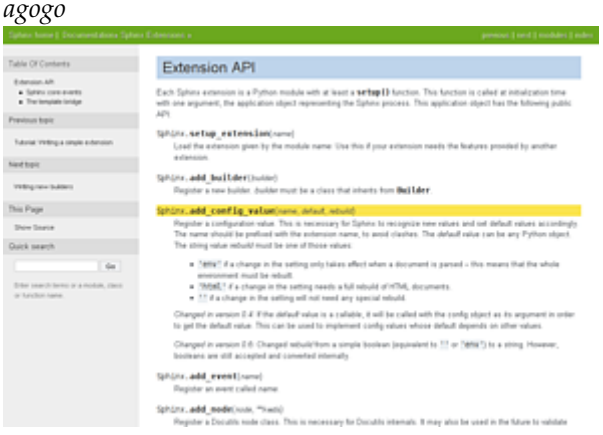
sphinxdoc

scrolls



agogo

traditional



nature

haiku

Sphinx comes with a selection of themes to choose from.

These themes are:

- **basic** – This is a basically unstyled layout used as the base for the other themes, and usable as the base for custom themes as well. The HTML contains all important elements like sidebar and relation bar. There is one option (which is inherited by the other themes):
  - **nosidebar** (true or false): Don't include the sidebar. Defaults to false.
- **default** – This is the default theme, which looks like the Python documentation. It can be customized via these options:
  - **rightsidebar** (true or false): Put the sidebar on the right side. Defaults to false.
  - **stickysidebar** (true or false): Make the sidebar "fixed" so that it doesn't scroll out of view for long body content. This may not work well with all browsers. Defaults to false.
  - **collapsiblesidebar** (true or false): Add an *experimental* JavaScript snippet that makes the sidebar collapsible via a button on its side. *Doesn't work together with "rightsidebar" or "stickysidebar".* Defaults to false.
  - **externalrefs** (true or false): Display external links differently from internal links. Defaults to false.

  There are also various color and font options that can change the color scheme without having to write a custom stylesheet:
  - **footerbgcolor** (CSS color): Background color for the footer line.
  - **footertextcolor** (CSS color): Text color for the footer line.
  - **sidebarbgcolor** (CSS color): Background color for the sidebar.
  - **sidebarbtncolor** (CSS color): Background color for the sidebar collapse button (used when *collapsiblesidebar* is true).
  - **sidebartextcolor** (CSS color): Text color for the sidebar.
  - **sidebarlinkcolor** (CSS color): Link color for the sidebar.
  - **relbarbgcolor** (CSS color): Background color for the relation bar.
  - **relbartextcolor** (CSS color): Text color for the relation bar.
  - **relbarlinkcolor** (CSS color): Link color for the relation bar.
  - **bgcolor** (CSS color): Body background color.
  - **textcolor** (CSS color): Body text color.
  - **linkcolor** (CSS color): Body link color.
  - **visitedlinkcolor** (CSS color): Body color for visited links.
  - **headbgcolor** (CSS color): Background color for headings.
  - **headtextcolor** (CSS color): Text color for headings.
  - **headlinkcolor** (CSS color): Link color for headings.
  - **codebgcolor** (CSS color): Background color for code blocks.
  - **codetextcolor** (CSS color): Default text color for code blocks, if not set differently by the highlighting style.
  - **bodyfont** (CSS font-family): Font for normal text.

  - **headfont** (CSS font-family): Font for headings.

- **sphinxdoc** – The theme used for this documentation. It features a sidebar on the right side. There are currently no options beyond *nosidebar*.
- **scrolls** – A more lightweight theme, based on the Jinja documentation. The following color options are available:
  - **headerbordercolor**
  - **subheadlinecolor**
  - **linkcolor**
  - **visitedlinkcolor**
  - **admonitioncolor**
- **agogo** – A theme created by Andi Albrecht. The following options are supported:
  - **bodyfont** (CSS font family): Font for normal text.
  - **headerfont** (CSS font family): Font for headings.
  - **pagewidth** (CSS length): Width of the page content, default 70em.
  - **documentwidth** (CSS length): Width of the document (without sidebar), default 50em.
  - **sidebarwidth** (CSS length): Width of the sidebar, default 20em.
  - **bgcolor** (CSS color): Background color.
  - **headerbg** (CSS value for "background"): background for the header area, default a grayish gradient.
  - **footerbg** (CSS value for "background"): background for the footer area, default a light gray gradient.
  - **linkcolor** (CSS color): Body link color.
  - **headercolor1**, **headercolor2** (CSS color): colors for <h1> and <h2> headings.
  - **headerlinkcolor** (CSS color): Color for the backreference link in headings.
  - **textalign** (CSS *text-align* value): Text alignment for the body, default is `justify`.
- **nature** – A greenish theme. There are currently no options beyond *nosidebar*.
- **haiku** – A theme without sidebar inspired by the Haiku OS user guide. The following options are supported:
  - **full_logo** (true or false, default false): If this is true, the header will only show the `html_logo`. Use this for large logos. If this is false, the logo (if present) will be shown floating right, and the documentation title will be put in the header.
  - **textcolor**, **headingcolor**, **linkcolor**, **visitedlinkcolor**, **hoverlinkcolor** (CSS colors): Colors for various body elements.
- **traditional** – A theme resembling the old Python documentation. There are currently no options beyond *nosidebar*.
- **epub** – A theme for the epub builder. There are currently no options. This theme tries to save visual space which is a sparse resource on ebook readers.

9.3 Creating themes

As said, themes are either a directory or a zipfile (whose name is the theme name), containing the following:

- A `theme.conf` file, see below.
- HTML templates, if needed.
- A `static/` directory containing any static files that will be copied to the output statid directory on build. These can be images, styles, script files.

The `theme.conf` file is in INI format [1] (readable by the standard Python `ConfigParser` module) and has the following structure:

```
[theme]
inherit = base theme
stylesheet = main CSS name
pygments_style = stylename

[options]
variable = default value
```

- The **inherit** setting gives the name of a "base theme", or `none`. The base theme will be used to locate missing templates (most themes will not have to supply most templates if they use `basic` as the base theme), its options will be inherited, and all of its static files will be used as well.
- The **stylesheet** setting gives the name of a CSS file which will be referenced in the HTML header. If you need more than one CSS file, either include one from the other via CSS' `@import`, or use a custom HTML template that adds `<link rel="stylesheet">` tags as necessary. Setting the `html_style` config value will override this setting.
- The **pygments_style** setting gives the name of a Pygments style to use for highlighting. This can be overridden by the user in the `pygments_style` config value.
- The **options** section contains pairs of variable names and default values. These options can be overridden by the user in `html_theme_options` and are accessible from all templates as `theme_<name>`.

9.3.1 Templating

The *guide to templating* is helpful if you want to write your own templates. What is important to keep in mind is the order in which Sphinx searches for templates:

- First, in the user's `templates_path` directories.
- Then, in the selected theme.
- Then, in its base theme, its base's base theme, etc.

When extending a template in the base theme with the same name, use the theme name as an explicit directory: `{% extends "basic/layout.html" %}`. From a user `templates_path` template, you can still use the "exclamation mark" syntax as described in the templating document.

9.3.2 Static templates

Since theme options are meant for the user to configure a theme more easily, without having to write a custom stylesheet, it is necessary to be able to template static files as well as HTML files. Therefore, Sphinx

[1] It is not an executable Python file, as opposed to `conf.py`, because that would pose an unnecessary security risk if themes are shared.

supports so-called "static templates", like this:

If the name of a file in the `static/` directory of a theme (or in the user's static path, for that matter) ends with `_t`, it will be processed by the template engine. The `_t` will be left from the final file name. For example, the *default* theme has a file `static/default.css_t` which uses templating to put the color options into the stylesheet. When a documentation is built with the default theme, the output directory will contain a `_static/default.css` file where all template tags have been processed.

CHAPTER

TEN

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.

10.1 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 post-process them using a custom tool, or use them in your Web application.

10.2 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 structure 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 selected theme's templates.

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' *basic* theme provides base templates with a couple of blocks it will fill with data. These are located in the `themes/basic` subdirectory of the Sphinx installation directory, and used by all builtin Sphinx themes. Templates with the same name in the `templates_path` override templates supplied by the selected theme.

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 overridden template with an exclamation mark, Sphinx will load the layout template from the underlying HTML theme.

Important: If you override a block, call `{{ super() }}` somewhere to render the block's content in the extended template – unless you don't want that content to show up.

10.3 Working with the builtin templates

The builtin **basic** theme supplies the templates that all builtin Sphinx themes are based on. It has the following elements you can override or use:

10.3.1 Blocks

The following blocks exist in the `layout.html` 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.

linktags 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 *relation bar*, 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 custom *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. It contains the block "body" where the individual content is put by subtemplates like `page.html`.

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.

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

The following four blocks are *only* used for pages that do not have assigned a list of custom sidebars in the `html_sidebars` config value. Their use is deprecated in favor of separate sidebar templates, which can be included via `html_sidebars`.

sidebartoc The table of contents within the sidebar. Deprecated since version 1.0.

sidebarrel The relation links (previous, next document) within the sidebar. Deprecated since version 1.0.

sidebarsourcelink The "Show source" link within the sidebar (normally only shown if this is enabled by `html_show_sourcelink`). Deprecated since version 1.0.

sidebarsearch The search box within the sidebar. Override this if you want to place some content at the bottom of the sidebar. Deprecated since version 1.0.

10.3.2 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;' %}
```

script_files
Add additional script files here, like this:

```
{% set script_files = script_files + ["_static/myscript.js"] %}
```

css_files
Similar to `script_files`, for CSS files.

10.3.3 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 relation bar.

10.3.4 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.

builder
The name of the builder (e.g. `html` or `htmlhelp`).

copyright
The value of `copyright`.

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

embedded
True if the built HTML is meant to be embedded in some viewing application that handles navigation, not the web browser, such as for HTML help or Qt help formats. In this case, the sidebar is not included.

favicon
The path to the HTML favicon in the static path, or ''.

file_suffix
The value of the builder's `out_suffix` attribute, i.e. the file name extension that the output files will get. For a standard HTML builder, this is usually `.html`.

has_source
True if the reST document sources are copied (if `html_copy_source` is true).

last_updated
The build date.

logo
The path to the HTML logo image in the static path, or ''.

master_doc
The value of `master_doc`, for usage with `pathto()`.

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 %}
```

pagename
The "page name" of the current file, i.e. either the document name if the file is generated from a reST source, or the equivalent hierarchical name relative to the output directory (`[directory/]filename_without_extension`).

parents
A list of parent documents for navigation, structured like the `next` item.

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

project
The value of `project`.

release
The value of `release`.

rellinks
A list of links to put at the left side of the relbar, next to "next" and "prev". This usually contains links to the general index and other indices, such as the Python module index. If you add something yourself, it must be a tuple `(pagename, link title, accesskey, link text)`.

shorttitle
The value of `html_short_title`.

show_source
True if `html_show_sourcelink` is true.

sphinx_version
The version of Sphinx used to build.

style
The name of the main stylesheet, as given by the theme or `html_style`.

title
The title of the current document, as used in the `<title>` tag.

use_opensearch
The value of `html_use_opensearch`.

version
The value of `version`.

In addition to these values, there are also all **theme options** available (prefixed by `theme_`), as well as the values given by the user in `html_context`.

In documents that are created from source files (as opposed to automatically-generated files like the module index, or documents that already are in HTML form), these variables are also available:

meta
Document metadata (a dictionary), see *File-wide metadata*.

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

toc
The local table of contents for the current page, rendered as HTML bullet lists.

toctree
A callable yielding the global TOC tree containing the current page, rendered as HTML bullet lists. Optional keyword arguments:

- `collapse` (true by default): if true, all TOC entries that are not ancestors of the current page are collapsed
- `maxdepth` (defaults to the max depth selected in the toctree directive): the maximum depth of the tree; set it to `-1` to allow unlimited depth
- `titles_only` (false by default): if true, put only toplevel document titles in the tree

CHAPTER

ELEVEN

SPHINX EXTENSIONS

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

This is what you can do in an extension: 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.

An extension is simply a Python module. When an extension is loaded, Sphinx imports this module and executes its `setup()` function, which in turn notifies Sphinx of everything the extension offers – see the extension tutorial for examples.

The configuration file itself can be treated as an extension if it contains a `setup()` function. All other extensions to load must be listed in the `extensions` configuration value.

11.1 Tutorial: Writing a simple extension

This section is intended as a walkthrough for the creation of custom extensions. It covers the basics of writing and activating an extensions, as well as commonly used features of extensions.

As an example, we will cover a "todo" extension that adds capabilities to include todo entries in the documentation, and collecting these in a central place. (A similar "todo" extension is distributed with Sphinx.)

11.1.1 Build Phases

One thing that is vital in order to understand extension mechanisms is the way in which a Sphinx project is built: this works in several phases.

Phase 0: Initialization

> In this phase, almost nothing interesting for us happens. The source directory is searched for source files, and extensions are initialized. Should a stored build environment exist, it is loaded, otherwise a new one is created.

Phase 1: Reading

> In Phase 1, all source files (and on subsequent builds, those that are new or changed) are read and parsed. This is the phase where directives and roles are encountered by the docutils, and the corresponding functions are called. The output of this phase is a *doctree* for each source files, that is a tree of docutils nodes. For document elements that aren't fully known until all existing files are read, temporary nodes are created.

During reading, the build environment is updated with all meta- and cross reference data of the read documents, such as labels, the names of headings, described Python objects and index entries. This will later be used to replace the temporary nodes.

The parsed doctrees are stored on the disk, because it is not possible to hold all of them in memory.

Phase 2: Consistency checks

Some checking is done to ensure no surprises in the built documents.

Phase 3: Resolving

Now that the metadata and cross-reference data of all existing documents is known, all temporary nodes are replaced by nodes that can be converted into output. For example, links are created for object references that exist, and simple literal nodes are created for those that don't.

Phase 4: Writing

This phase converts the resolved doctrees to the desired output format, such as HTML or LaTeX. This happens via a so-called docutils writer that visits the individual nodes of each doctree and produces some output in the process.

Note: Some builders deviate from this general build plan, for example, the builder that checks external links does not need anything more than the parsed doctrees and therefore does not have phases 2–4.

11.1.2 Extension Design

We want the extension to add the following to Sphinx:

- A "todo" directive, containing some content that is marked with "TODO", and only shown in the output if a new config value is set. (Todo entries should not be in the output by default.)
- A "todolist" directive that creates a list of all todo entries throughout the documentation.

For that, we will need to add the following elements to Sphinx:

- New directives, called `todo` and `todolist`.
- New document tree nodes to represent these directives, conventionally also called `todo` and `todolist`. We wouldn't need new nodes if the new directives only produced some content representable by existing nodes.
- A new config value `todo_include_todos` (config value names should start with the extension name, in order to stay unique) that controls whether todo entries make it into the output.
- New event handlers: one for the `doctree-resolved` event, to replace the todo and todolist nodes, and one for `env-purge-doc` (the reason for that will be covered later).

11.1.3 The Setup Function

The new elements are added in the extension's setup function. Let us create a new Python module called `todo.py` and add the setup function:

```
def setup(app):
    app.add_config_value('todo_include_todos', False, False)

    app.add_node(todolist)
    app.add_node(todo,
```

```
                  html=(visit_todo_node, depart_todo_node),
                  latex=(visit_todo_node, depart_todo_node),
                  text=(visit_todo_node, depart_todo_node))

    app.add_directive('todo', TodoDirective)
    app.add_directive('todolist', TodolistDirective)
    app.connect('doctree-resolved', process_todo_nodes)
    app.connect('env-purge-doc', purge_todos)
```

The calls in this function refer to classes and functions not yet written. What the individual calls do is the following:

- `add_config_value()` lets Sphinx know that it should recognize the new *config value* `todo_include_todos`, whose default value should be `False` (this also tells Sphinx that it is a boolean value).

  If the third argument was `True`, all documents would be re-read if the config value changed its value. This is needed for config values that influence reading (build phase 1).

- `add_node()` adds a new *node class* to the build system. It also can specify visitor functions for each supported output format. These visitor functions are needed when the new nodes stay until phase 4 – since the `todolist` node is always replaced in phase 3, it doesn't need any.

  We need to create the two node classes `todo` and `todolist` later.

- `add_directive()` adds a new *directive*, given by name and class.

  The handler functions are created later.

- Finally, `connect()` adds an *event handler* to the event whose name is given by the first argument. The event handler function is called with several arguments which are documented with the event.

11.1.4 The Node Classes

Let's start with the node classes:

```
from docutils import nodes

class todo(nodes.Admonition, nodes.Element):
    pass

class todolist(nodes.General, nodes.Element):
    pass

def visit_todo_node(self, node):
    self.visit_admonition(node)

def depart_todo_node(self, node):
    self.depart_admonition(node)
```

Node classes usually don't have to do anything except inherit from the standard docutils classes defined in `docutils.nodes`. `todo` inherits from `Admonition` because it should be handled like a note or warning, `todolist` is just a "general" node.

11.1.5 The Directive Classes

A directive class is a class deriving usually from `docutils.parsers.rst.Directive`. Since the class-based directive interface doesn't exist yet in Docutils 0.4, Sphinx has another base class called

`sphinx.util.compat.Directive` that you can derive your directive from, and it will work with both Docutils 0.4 and 0.5 upwards. The directive interface is covered in detail in the docutils documentation; the important thing is that the class has a method `run` that returns a list of nodes.

The `todolist` directive is quite simple:

```
from sphinx.util.compat import Directive

class TodolistDirective(Directive):

    def run(self):
        return [todolist('')]
```

An instance of our `todolist` node class is created and returned. The todolist directive has neither content nor arguments that need to be handled.

The `todo` directive function looks like this:

```
from sphinx.util.compat import make_admonition

class TodoDirective(Directive):

    # this enables content in the directive
    has_content = True

    def run(self):
        env = self.state.document.settings.env

        targetid = "todo-%d" % env.new_serialno('todo')
        targetnode = nodes.target('', '', ids=[targetid])

        ad = make_admonition(todo, self.name, [_('Todo')], self.options,
                             self.content, self.lineno, self.content_offset,
                             self.block_text, self.state, self.state_machine)

        if not hasattr(env, 'todo_all_todos'):
            env.todo_all_todos = []
        env.todo_all_todos.append({
            'docname': env.docname,
            'lineno': self.lineno,
            'todo': ad[0].deepcopy(),
            'target': targetnode,
        })

        return [targetnode] + ad
```

Several important things are covered here. First, as you can see, you can refer to the build environment instance using `self.state.document.settings.env`.

Then, to act as a link target (from the todolist), the todo directive needs to return a target node in addition to the todo node. The target ID (in HTML, this will be the anchor name) is generated by using `env.new_serialno` which is returns a new integer directive on each call and therefore leads to unique target names. The target node is instantiated without any text (the first two arguments).

An admonition is created using a standard docutils function (wrapped in Sphinx for docutils cross-version compatibility). The first argument gives the node class, in our case `todo`. The third argument gives the admonition title (use `arguments` here to let the user specify the title). A list of nodes is returned from `make_admonition`.

Then, the todo node is added to the environment. This is needed to be able to create a list of all todo entries

throughout the documentation, in the place where the author puts a `todolist` directive. For this case, the environment attribute `todo_all_todos` is used (again, the name should be unique, so it is prefixed by the extension name). It does not exist when a new environment is created, so the directive must check and create it if necessary. Various information about the todo entry's location are stored along with a copy of the node.

In the last line, the nodes that should be put into the doctree are returned: the target node and the admonition node.

The node structure that the directive returns looks like this:

```
+--------------------+
| target node        |
+--------------------+
+--------------------+
| todo node          |
+--------------------+
  \__+--------------------+
     | admonition title   |
     +--------------------+
     | paragraph          |
     +--------------------+
     | ...                |
     +--------------------+
```

11.1.6 The Event Handlers

Finally, let's look at the event handlers. First, the one for the `env-purge-doc` event:

```
def purge_todos(app, env, docname):
    if not hasattr(env, 'todo_all_todos'):
        return
    env.todo_all_todos = [todo for todo in env.todo_all_todos
                          if todo['docname'] != docname]
```

Since we store information from source files in the environment, which is persistent, it may become out of date when the source file changes. Therefore, before each source file is read, the environment's records of it are cleared, and the `env-purge-doc` event gives extensions a chance to do the same. Here we clear out all todos whose docname matches the given one from the `todo_all_todos` list. If there are todos left in the document, they will be added again during parsing.

The other handler belongs to the `doctree-resolved` event. This event is emitted at the end of phase 3 and allows custom resolving to be done:

```
def process_todo_nodes(app, doctree, fromdocname):
    if not app.config.todo_include_todos:
        for node in doctree.traverse(todo):
            node.parent.remove(node)

    # Replace all todolist nodes with a list of the collected todos.
    # Augment each todo with a backlink to the original location.
    env = app.builder.env

    for node in doctree.traverse(todolist):
        if not app.config.todo_include_todos:
            node.replace_self([])
            continue
```

```
    content = []

    for todo_info in env.todo_all_todos:
        para = nodes.paragraph()
        filename = env.doc2path(todo_info['docname'], base=None)
        description = (
            _('(The original entry is located in %s, line %d and can be found ') %
            (filename, todo_info['lineno']))
        para += nodes.Text(description, description)

        # Create a reference
        newnode = nodes.reference('', '')
        innernode = nodes.emphasis(_('here'), _('here'))
        newnode['refdocname'] = todo_info['docname']
        newnode['refuri'] = app.builder.get_relative_uri(
            fromdocname, todo_info['docname'])
        newnode['refuri'] += '#' + todo_info['target']['refid']
        newnode.append(innernode)
        para += newnode
        para += nodes.Text('.)', '.)')

        # Insert into the todolist
        content.append(todo_info['todo'])
        content.append(para)

    node.replace_self(content)
```

It is a bit more involved. If our new "todo_include_todos" config value is false, all todo and todolist nodes are removed from the documents.

If not, todo nodes just stay where and how they are. Todolist nodes are replaced by a list of todo entries, complete with backlinks to the location where they come from. The list items are composed of the nodes from the todo entry and docutils nodes created on the fly: a paragraph for each entry, containing text that gives the location, and a link (reference node containing an italic node) with the backreference. The reference URI is built by `app.builder.get_relative_uri` which creates a suitable URI depending on the used builder, and appending the todo node's (the target's) ID as the anchor name.

11.2 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:

`Sphinx.`**`setup_extension`**`(`*name*`)`
Load the extension given by the module *name*. Use this if your extension needs the features provided by another extension.

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

`Sphinx.`**`add_config_value`**`(`*name*, *default*, *rebuild*`)`
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 string value *rebuild* must be one of those values:

- `'env'` if a change in the setting only takes effect when a document is parsed – this means that the whole environment must be rebuilt.

- `'html'` if a change in the setting needs a full rebuild of HTML documents.
- `''` if a change in the setting will not need any special rebuild.

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.Changed in version 0.6: Changed *rebuild* from a simple boolean (equivalent to `''` or `'env'`) to a string. However, booleans are still accepted and converted internally.

`Sphinx.`**`add_domain`**(*domain*)
Make the given *domain* (which must be a class; more precisely, a subclass of `Domain`) known to Sphinx. New in version 1.0.

`Sphinx.`**`override_domain`**(*domain*)
Make the given *domain* class known to Sphinx, assuming that there is already a domain with its `.name`. The new domain must be a subclass of the existing one. New in version 1.0.

`Sphinx.`**`add_index_to_domain`**(*domain*, *index*)
Add a custom *index* class to the domain named *domain*. *index* must be a subclass of `Index`. New in version 1.0.

`Sphinx.`**`add_event`**(*name*)
Register an event called *name*. This is needed to be able to emit it.

`Sphinx.`**`add_node`**(*node*, ***kwds*)
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.

Node visitor functions for the Sphinx HTML, LaTeX, text and manpage writers can be given as keyword arguments: the keyword must be one or more of `'html'`, `'latex'`, `'text'`, `'man'`, the value a 2-tuple of `(visit, depart)` methods. `depart` can be `None` if the `visit` function raises `docutils.nodes.SkipNode`. Example:

```
class math(docutils.nodes.Element): pass

def visit_math_html(self, node):
    self.body.append(self.starttag(node, 'math'))
def depart_math_html(self, node):
    self.body.append('</math>')

app.add_node(math, html=(visit_math_html, depart_math_html))
```

Obviously, translators for which you don't specify visitor methods will choke on the node when encountered in a document to translate. Changed in version 0.5: Added the support for keyword arguments giving visit functions.

`Sphinx.`**`add_directive`**(*name*, *func*, *content*, *arguments*, ***options*)
`Sphinx.`**`add_directive`**(*name*, *directiveclass*)
Register a Docutils directive. *name* must be the prospective directive name. There are two possible ways to write a directive:

- In the docutils 0.4 style, *obj* is the directive function. *content*, *arguments* and *options* are set as attributes on the function and determine whether the directive has content, arguments and options, respectively. **This style is deprecated.**
- In the docutils 0.5 style, *directiveclass* is the directive class. It must already have attributes named *has_content*, *required_arguments*, *optional_arguments*, *final_argument_whitespace* and *option_spec* that correspond to the options for the function way. See the Docutils docs for details.

  The directive class normally must inherit from the class `docutils.parsers.rst.Directive`. When writing a directive for usage in a Sphinx

extension, you inherit from `sphinx.util.compat.Directive` instead which does the right thing even on docutils 0.4 (which doesn't support directive classes otherwise).

For example, the (already existing) `literalinclude` directive would be added like this:

```
from docutils.parsers.rst import directives
add_directive('literalinclude', literalinclude_directive,
              content = 0, arguments = (1, 0, 0),
              linenos = directives.flag,
              language = direcitves.unchanged,
              encoding = directives.encoding)
```

Changed in version 0.6: Docutils 0.5-style directive classes are now supported.

Sphinx.**add_directive_to_domain**(*domain*, *name*, *func*, *content*, *arguments*, ***options*)
Sphinx.**add_directive_to_domain**(*domain*, *name*, *directiveclass*)
Like `add_directive()`, but the directive is added to the domain named *domain*. New in version 1.0.

Sphinx.**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).

Sphinx.**add_role_to_domain**(*domain*, *name*, *role*)
Like `add_role()`, but the role is added to the domain named *domain*. New in version 1.0.

Sphinx.**add_generic_role**(*name*, *nodeclass*)
Register a Docutils role that does nothing but wrap its contents in the node given by *nodeclass*. New in version 0.6.

Sphinx.**add_object_type**(*directivename*, *rolename*, *indextemplate=''*, *parse_node=None*, *ref_nodeclass=None*, *objname=''*, *doc_field_types=[]*)
This method is a very convenient way to add a new *object* type that can be cross-referenced. It will do this:

- Create a new directive (called *directivename*) for documenting an object. 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 object descriptions.
- 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 `conf.py` file in the source for this documentation for an example.
- The *objname* (if not given, will default to *directivename*) names the type of object. It is used when listing objects, e.g. in search results.

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

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

you can use this markup in your documents:

```
.. rst:directive:: function

   Document a function.

<...>

See also the :rst: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 syntactical possibilities as for standard Sphinx roles (see *Cross-referencing syntax*).

This method is also available under the deprecated alias `add_description_unit`.

`Sphinx.`**`add_crossref_type`**(*directivename*, *rolename*, *indextemplate=''*, *ref_nodeclass=None*, *objname=''*)

This method is very similar to `add_object_type()` 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.)

`Sphinx.`**`add_transform`**(*transform*)

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

`Sphinx.`**`add_javascript`**(*filename*)

Add *filename* to the list of JavaScript files that the default HTML template will include. The filename must be relative to the HTML static path, see `the docs for the config value`. A full URI with scheme, like `http://example.org/foo.js`, is also supported. New in version 0.5.

`Sphinx.`**`add_stylesheet`**(*filename*)

Add *filename* to the list of CSS files that the default HTML template will include. Like for `add_javascript()`, the filename must be relative to the HTML static path. New in version 1.0.

`Sphinx.`**`add_lexer`**(*alias*, *lexer*)

Use *lexer*, which must be an instance of a Pygments lexer class, to highlight code blocks with the given language *alias*. New in version 0.6.

`Sphinx.`**`add_autodocumenter`**(*cls*)

Add *cls* as a new documenter class for the `sphinx.ext.autodoc` extension. It must be a subclass of `sphinx.ext.autodoc.Documenter`. This allows to auto-document new types of objects. See the source of the autodoc module for examples on how to subclass `Documenter`. New in version 0.6.

`Sphinx.`**`add_autodoc_attrgetter`**(*type*, *getter*)

Add *getter*, which must be a function with an interface compatible to the `getattr()` builtin, as the

autodoc attribute getter for objects that are instances of *type*. All cases where autodoc needs to get an attribute of a type are then handled by this function instead of `getattr()`. New in version 0.6.

`Sphinx.`**`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()`.

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

`Sphinx.`**`emit`**(*event*, **arguments*)
Emit *event* and pass *arguments* to the callback functions. Return the return values of all callbacks as a list. Do not emit core Sphinx events in extensions!

`Sphinx.`**`emit_firstresult`**(*event*, **arguments*)
Emit *event* and pass *arguments* to the callback functions. Return the result of the first callback that doesn't return `None`. New in version 0.5.

`Sphinx.`**`require_sphinx`**(*version*)
Compare *version* (which must be a `major.minor` version string, e.g. `'1.1'`) with the version of the running Sphinx, and abort the build when it is too old. New in version 1.0.

exception `sphinx.application.`**`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.

11.2.1 Sphinx core events

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

`builder-inited`(*app*)
Emitted when the builder object has been created. It is available as `app.builder`.

`env-purge-doc`(*app*, *env*, *docname*)
Emitted when all traces of a source file should be cleaned from the environment, that is, if the source file is removed or before it is freshly read. This is for extensions that keep their own caches in attributes of the environment.

For example, there is a cache of all modules on the environment. When a source file has been changed, the cache's entries for the file are cleared, since the module declarations could have been removed from the file. New in version 0.5.

`source-read`(*app*, *docname*, *source*)
Emitted when a source file has been read. The *source* argument is a list whose single element is the contents of the source file. You can process the contents and replace this item to implement source-level transformations.

For example, if you want to use `$` signs to delimit inline math, like in LaTeX, you can use a regular expression to replace `$...$` by `:math:`...``. New in version 0.5.

`doctree-read`(*app*, *doctree*)
Emitted when a doctree has been parsed and read by the environment, and is about to be pickled. The *doctree* can be modified in-place.

`missing-reference`(*app*, *env*, *node*, *contnode*)
Emitted when a cross-reference to a Python module or object cannot be resolved. If the event handler can resolve the reference, it should return a new docutils node to be inserted in the document tree in place of the node *node*. Usually this node is a `reference` node containing *contnode* as a child.

Parameters

- **env** – The build environment (`app.builder.env`).
- **node** – The `pending_xref` node to be resolved. Its attributes `reftype`, `reftarget`, `modname` and `classname` attributes determine the type and target of the reference.
- **contnode** – The node that carries the text and formatting inside the future reference and should be a child of the returned reference node.

New in version 0.5.

doctree-resolved(*app*, *doctree*, *docname*)
Emitted when a doctree has been "resolved" by the environment, that is, all references have been resolved and TOCs have been inserted. The *doctree* can be modified in place.

Here is the place to replace custom nodes that don't have visitor methods in the writers, so that they don't cause errors when the writers encounter them.

env-updated(*app*, *env*)
Emitted when the `update()` method of the build environment has completed, that is, the environment and all doctrees are now up-to-date. New in version 0.5.

html-collect-pages(*app*)
Emitted when the HTML builder is starting to write non-document pages. You can add pages to write by returning an iterable from this event consisting of `(pagename, context, templatename)`. New in version 1.0.

html-page-context(*app*, *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.

build-finished(*app*, *exception*)
Emitted when a build has finished, before Sphinx exits, usually used for cleanup. This event is emitted even when the build process raised an exception, given as the *exception* argument. The exception is reraised in the application after the event handlers have run. If the build process raised no exception, *exception* will be `None`. This allows to customize cleanup actions depending on the exception status. New in version 0.5.

11.2.2 The template bridge

class `sphinx.application.`**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*, *theme=None*, *dirs=None*)
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`.

theme is a `sphinx.theming.Theme` object or None; in the latter case, *dirs* can be list of fixed directories to look for templates.

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 given as a filename with a specified context (a Python dictionary).

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

11.2.3 Domain API

class sphinx.domains.**Domain**(*env*)
A Domain is meant to be a group of "object" description directives for objects of a similar nature, and corresponding roles to create references to them. Examples would be Python modules, classes, functions etc., elements of a templating language, Sphinx roles and directives, etc.

Each domain has a separate storage for information about existing objects and how to reference them in *self.data*, which must be a dictionary. It also must implement several functions that expose the object information in a uniform way to parts of Sphinx that allow the user to reference or search for objects in a domain-agnostic way.

About *self.data*: since all object and cross-referencing information is stored on a BuildEnvironment instance, the *domain.data* object is also stored in the *env.domaindata* dict under the key *domain.name*. Before the build process starts, every active domain is instantiated and given the environment object; the *domaindata* dict must then either be nonexistent or a dictionary whose 'version' key is equal to the domain class' `data_version` attribute. Otherwise, *IOError* is raised and the pickled environment is discarded.

clear_doc(*docname*)
Remove traces of a document in the domain-specific inventories.

directive(*name*)
Return a directive adapter class that always gives the registered directive its full name ('domain:name') as `self.name`.

get_objects()
Return an iterable of "object descriptions", which are tuples with five items:

- *name* – fully qualified name
- *dispname* – name to display when searching/linking
- *type* – object type, a key in `self.object_types`
- *docname* – the document where it is to be found
- *anchor* – the anchor name for the object
- *priority* – how "important" the object is (determines placement in search results)
  - –1: default priority (placed before full-text matches)
  - –0: object is important (placed before default-priority objects)
  - –2: object is unimportant (placed after full-text matches)

—-1: object should not show up in search at all

get_type_name(*type*, *primary=False*)
Return full name for given ObjType.

process_doc(*env*, *docname*, *document*)
Process a document after it is read by the environment.

resolve_xref(*env*, *fromdocname*, *builder*, *typ*, *target*, *node*, *contnode*)
Resolve the `pending_xref` *node* with the given *typ* and *target*.

This method should return a new node, to replace the xref node, containing the *contnode* which is the markup content of the cross-reference.

If no resolution can be found, None can be returned; the xref node will then given to the 'missing-reference' event, and if that yields no resolution, replaced by *contnode*.

The method can also raise `sphinx.environment.NoUri` to suppress the 'missing-reference' event being emitted.

role(*name*)
Return a role adapter function that always gives the registered role its full name ('domain:name') as the first argument.

dangling_warnings
role name -> a warning message if reference is missing

data_version
data version, bump this when the format of *self.data* changes

directives
directive name -> directive class

indices
a list of Index subclasses

initial_data
data value for a fresh environment

label
domain label: longer, more descriptive (used in messages)

name
domain name: should be short, but unique

object_types
type (usually directive) name -> ObjType instance

roles
role name -> role callable

class `sphinx.domains.`**ObjType**(*lname*, **roles*, ***attrs*)
An ObjType is the description for a type of object that a domain can document. In the object_types attribute of Domain subclasses, object type names are mapped to instances of this class.

Constructor arguments:

- *lname*: localized name of the type (do not include domain name)
- *roles*: all the roles that can refer to an object of this type
- *attrs*: object attributes – currently only "searchprio" is known, which defines the object's priority in the full-text search index, see `Domain.get_objects()`.

class sphinx.domains.**Index**(*domain*)
An Index is the description for a domain-specific index. To add an index to a domain, subclass Index, overriding the three name attributes:

- *name* is an identifier used for generating file names.
- *localname* is the section title for the index.
- *shortname* is a short name for the index, for use in the relation bar in HTML output. Can be empty to disable entries in the relation bar.

and providing a `generate()` method. Then, add the index class to your domain's *indices* list. Extensions can add indices to existing domains using `add_index_to_domain()`.

generate(*docnames=None*)
Return entries for the index given by *name*. If *docnames* is given, restrict to entries referring to these docnames.

The return value is a tuple of `(content, collapse)`, where *collapse* is a boolean that determines if sub-entries should start collapsed (for output formats that support collapsing sub-entries).

content is a sequence of `(letter, entries)` tuples, where *letter* is the "heading" for the given *entries*, usually the starting letter.

entries is a sequence of single entries, where a single entry is a sequence `[name, subtype, docname, anchor, extra, qualifier, descr]`. The items in this sequence have the following meaning:

- *name* – the name of the index entry to be displayed
- *subtype* – sub-entry related type: 0 – normal entry 1 – entry with sub-entries 2 – sub-entry
- *docname* – docname where the entry is located
- *anchor* – anchor for the entry within *docname*
- *extra* – extra info for the entry
- *qualifier* – qualifier for the description
- *descr* – description for the entry

Qualifier and description are not rendered e.g. in LaTeX output.

11.3 Writing new builders

Todo

Expand this.

class sphinx.builders.**Builder**
This is the base class for all builders.

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

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 *filenames*.

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

build(*docnames*, *summary=None*, *method='update'*)
Main build method. First updates the environment, and then calls `write()`.

These methods can be overridden in concrete builder classes:

init()
Load necessary templates and perform initialization. The default implementation does nothing.

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

If the builder does not output individual files corresponding to source files, return a string here. If it does, return an iterable of those files that need to be written.

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()
Finish the building process. The default implementation does nothing.

11.4 Builtin Sphinx extensions

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

11.4.1 `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.

Note: For Sphinx (actually, the Python interpreter that executes Sphinx) to find your module, it must be importable. That means that the module or the package must be in one of the directories on `sys.path` – adapt your `sys.path` in the configuration file accordingly.

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 `py:module`, `py: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 `py:module`, `py:class` etc. directive.

Note: Just as `py:class` respects the current `py:module`, `autoclass` will also do so. Likewise, `automethod` will respect the current `py: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:

  ```
  .. automodule:: noodle
     :members:
  ```

  will document all module members (recursively), and

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

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

  For modules, `__all__` will be respected when looking for members; the order of the members will also be the order in `__all__`.

  You can also give an explicit list of members; only these will then be documented:

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

- If you want to make the `members` option the default, see `autodoc_default_flags`.
- Members without docstrings will be left out, unless you give the `undoc-members` flag option:

  ```
  .. automodule:: noodle
     :members:
     :undoc-members:
  ```

- For classes and exceptions, members inherited from base classes will be left out when documenting all members, 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.

  Note: this will lead to markup errors if the inherited members come from a module whose docstrings are not reST formatted. New in version 0.3.

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

  ```
  .. 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 `automodule`, `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 (when used with `automodule`, this will be inserted for every class that is documented in the module). New in version 0.4.

- All autodoc directives support the `noindex` flag option that has the same effect as for standard `py:function` etc. directives: no index entries are generated for the documented object (and all autodocumented members). New in version 0.4.

- `automodule` also recognizes the `synopsis`, `platform` and `deprecated` options that the standard `py:module` directive supports. New in version 0.5.

- `automodule` and `autoclass` also has an `member-order` option that can be used to override the global value of `autodoc_member_order` for one directive. New in version 0.6.

- The directives supporting member documentation also have a `exclude-members` option that can be used to exclude single member names from documentation, if all members are to be documented. New in version 0.6.

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::
.. autodata::
.. automethod::
.. autoattribute::
```

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

For module data members and class attributes, documentation can either be put into a special-formatted comment *before* the attribute definition, or in a docstring *after* the definition. This means that in the following class definition, both attributes can be autodocumented:

```
class Foo:
    """Docstring for class Foo."""
```

```
#: Doc comment for attribute Foo.bar.
bar = 1

baz = 2
"""Docstring for attribute Foo.baz."""
```

Changed in version 0.6: autodata and autoattribute can now extract docstrings.

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:

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.

autodoc_member_order
This value selects if automatically documented members are sorted alphabetical (value 'alphabetical'), by member type (value 'groupwise') or by source order (value 'bysource'). The default is alphabetical.

Note that for source order, the module must be a Python module with the source code available. New in version 0.6.Changed in version 1.0: Support for 'bysource'.

autodoc_default_flags
This value is a list of autodoc directive flags that should be automatically applied to all autodoc directives. The supported flags are 'members','undoc-members','inherited-members' and 'show-inheritance'.

If you set one of these flags in this config value, you can use a negated form, ''no-flag'', in an autodoc directive, to disable it once. For example, if autodoc_default_flags is set to ['members', 'undoc-members'], and you write a directive like this:

```
.. automodule:: foo
   :no-undoc-members:
```

the directive will be interpreted as if only :members: was given. New in version 1.0.

Docstring preprocessing

autodoc provides the following additional events:

autodoc-process-docstring(*app*, *what*, *name*, *obj*, *options*, *lines*)
New in version 0.4. 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

autodoc-process-signature(*app*, *what*, *name*, *obj*, *options*, *signature*, *return_annotation*)

New in version 0.5. Emitted when autodoc has formatted a signature for an object. The event handler can return a new tuple `(signature, return_annotation)` 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
- **signature** – function signature, as a string of the form `"(parameter_1, parameter_2)"`, or `None` if introspection didn't succeed and signature wasn't specified in the directive.
- **return_annotation** – function return annotation as a string of the form `" -> annotation"`, or `None` if there is no return annotation

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

`sphinx.ext.autodoc.`**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`.

`sphinx.ext.autodoc.`**between**(*marker*, *what=None*, *keepempty=False*, *exclude=False*)

Return a listener that either keeps, or if *exclude* is True excludes, 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.

Skipping members

autodoc allows the user to define a custom method for determining whether a member should be included in the documentation by using the following event:

autodoc-skip-member (*app*, *what*, *name*, *obj*, *skip*, *options*)
New in version 0.5. Emitted when autodoc has to decide whether a member should be included in the documentation. The member is excluded if a handler returns `True`. It is included if the handler returns `False`.

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
- **skip** – a boolean indicating if autodoc will skip this member if the user handler does not override the decision
- **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

11.4.2 `sphinx.ext.autosummary` – Generate autodoc summaries

New in version 0.6. This extension generates function/method/attribute summary lists, similar to those output e.g. by Epydoc and other API doc generation tools. This is especially useful when your docstrings are long and detailed, and putting each one of them on a separate page makes them easier to read.

The `sphinx.ext.autosummary` extension does this in two parts:

1. There is an `autosummary` directive for generating summary listings that contain links to the documented items, and short summary blurbs extracted from their docstrings.
2. The convenience script **sphinx-autogen** or the new `autosummary_generate` config value can be used to generate short "stub" files for the entries listed in the `autosummary` directives. These by default contain only the corresponding `sphinx.ext.autodoc` directive.

.. autosummary::
Insert a table that contains links to documented items, and a short summary blurb (the first sentence of the docstring) for each of them. The `autosummary` directive can also optionally serve as a `toctree` entry for the included items.

For example,

```
.. currentmodule:: sphinx

.. autosummary::

   environment.BuildEnvironment
   util.relative_uri
```

produces a table like this:

environment.BuildEnvironment(srcdir, ...)	The environment in which the ReST files are translated.
util.relative_uri(base, to)	Return a relative URL from base to to.

Autosummary preprocesses the docstrings and signatures with the same autodoc-process-docstring and autodoc-process-signature hooks as autodoc.

Options

- If you want the autosummary table to also serve as a toctree entry, use the toctree option, for example:

```
.. autosummary::
   :toctree: DIRNAME

   sphinx.environment.BuildEnvironment
   sphinx.util.relative_uri
```

  The toctree option also signals to the **sphinx-autogen** script that stub pages should be generated for the entries listed in this directive. The option accepts a directory name as an argument; **sphinx-autogen** will by default place its output in this directory. If no argument is given, output is placed in the same directory as the file that contains the directive.

- If you don't want the autosummary to show function signatures in the listing, include the nosignatures option:

```
.. autosummary::
   :nosignatures:

   sphinx.environment.BuildEnvironment
   sphinx.util.relative_uri
```

- You can specify a custom template with the template option. For example,

```
.. autosummary::
   :template: mytemplate.rst

   sphinx.environment.BuildEnvironment
```

  would use the template mytemplate.rst in your templates_path to generate the pages for all entries listed. See Customizing templates below. New in version 1.0.

sphinx-autogen – generate autodoc stub pages

The **sphinx-autogen** script can be used to conveniently generate stub documentation pages for items included in autosummary listings.

For example, the command

```
$ sphinx-autogen -o generated *.rst
```

will read all autosummary tables in the *.rst files that have the :toctree: option set, and output corresponding stub pages in directory generated for all documented items. The generated pages by default contain text of the form:

```
sphinx.util.relative_uri
========================

.. autofunction:: sphinx.util.relative_uri
```

If the `-o` option is not given, the script will place the output files in the directories specified in the `:toctree:` options.

Generating stub pages automatically

If you do not want to create stub pages with **sphinx-autogen**, you can also use this new config value:

autosummary_generate
: Boolean indicating whether to scan all found documents for autosummary directives, and to generate stub pages for each.

  Can also be a list of documents for which stub pages should be generated.

  The new files will be placed in the directories specified in the `:toctree:` options of the directives.

Customizing templates

New in version 1.0. You can customize the stub page templates, in a similar way as the HTML Jinja templates, see *Templating*. (`TemplateBridge` is not supported.)

Note: If you find yourself spending much time tailoring the stub templates, this may indicate that it's a better idea to write custom narrative documentation instead.

Autosummary uses the following template files:

- `autosummary/base.rst` – fallback template
- `autosummary/module.rst` – template for modules
- `autosummary/class.rst` – template for classes
- `autosummary/function.rst` – template for functions
- `autosummary/attribute.rst` – template for class attributes
- `autosummary/method.rst` – template for class methods

The following variables available in the templates:

name
: Name of the documented object, excluding the module and class parts.

objname
: Name of the documented object, excluding the module parts.

fullname
: Full name of the documented object, including module and class parts.

module
: Name of the module the documented object belongs to.

class
: Name of the class the documented object belongs to. Only available for methods and attributes.

underline
A string containing `len(full_name) * '='`.

members
List containing names of all members of the module or class. Only available for modules and classes.

functions
List containing names of "public" functions in the module. Here, "public" here means that the name does not start with an underscore. Only available for modules.

classes
List containing names of "public" classes in the module. Only available for modules.

exceptions
List containing names of "public" exceptions in the module. Only available for modules.

methods
List containing names of "public" methods in the class. Only available for classes.

attributes
List containing names of "public" attributes in the class. Only available for classes.

Note: You can use the `autosummary` directive in the stub pages. Stub pages are generated also based on these directives.

11.4.3 `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.

Note: Code in a testcode block is always executed all at once, no matter how many statements it contains. Therefore, output will *not* be generated for bare expressions – use print. Example:

```
.. testcode::

   1+1        # this will give no output!
   print 2+2  # this will give output

.. testoutput::

   4
```

Also, please be aware that since the doctest module does not support mixing regular output and an exception message in the same snippet, this applies to testcode/testoutput as well.

.. **testoutput**:: [group]

The corresponding output, or the exception message, 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:

```
.. testcode::

   print 'Output     text.'
```

```
   .. 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 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_global_setup
: Python code that is treated like it were put in a `testsetup` directive for *every* file that is tested, and for every group. You can use this to e.g. import modules you will always need in your doctests. New in version 0.6.

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 used to introduce a doctest block; docutils recognizes them from the leading `>>>`. Also, no additional indentation is used, 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, though you may set `trim_doctest_flags` to achieve the latter in all code blocks with Python console content.

11.4.4 `sphinx.ext.intersphinx` – Link to other projects' documentation

New in version 0.5. This extension can generate automatic links to the documentation of objects in other projects.

Usage is simple: whenever Sphinx encounters a cross-reference that has no matching target in the current documentation set, it looks for targets in the documentation sets configured in `intersphinx_mapping`. A reference like `:py:class:`zipfile.ZipFile`` can then link to the Python documentation for the ZipFile class, without you having to specify where it is located exactly.

When using the "new" format (see below), you can even force lookup in a foreign set by prefixing the link target appropriately. A link like `:ref:`comparison manual <python:comparisons>`` will then link to the label "comparisons" in the doc set "python", if it exists.

Behind the scenes, this works as follows:

- Each Sphinx HTML build creates a file named `objects.inv` that contains a mapping from object names to URIs relative to the HTML set's root.
- Projects using the Intersphinx extension can specify the location of such mapping files in the `intersphinx_mapping` config value. The mapping will then be used to resolve otherwise missing references to objects into links to the other documentation.
- By default, the mapping file is assumed to be at the same location as the rest of the documentation; however, the location of the mapping file can also be specified individually, e.g. if the docs should be buildable without Internet access.

To use intersphinx linking, add `'sphinx.ext.intersphinx'` to your `extensions` config value, and use these new config values to activate linking:

`intersphinx_mapping`
This config value contains the locations and names of other projects that should be linked to in this documentation.

Relative local paths for target locations are taken as relative to the base of the built documentation, while relative local paths for inventory locations are taken as relative to the source directory.

When fetching remote inventory files, proxy settings will be read from the `$HTTP_PROXY` environment variable.

Old format for this config value

This is the format used before Sphinx 1.0. It is still recognized.

A dictionary mapping URIs to either `None` or an URI. The keys are the base URI of the foreign Sphinx documentation sets and can be local paths or HTTP URIs. The values indicate where the inventory file can be found: they can be `None` (at the same location as the base URI) or another local or HTTP URI.

New format for this config value New in version 1.0. A dictionary mapping unique identifiers to a tuple `(target, inventory)`. Each `target` is the base URI of a foreign Sphinx documentation set and can be a local path or an HTTP URI. The `inventory` indicates where the inventory file can be found: it can be `None` (at the same location as the base URI) or another local or HTTP URI.

The unique identifier can be used to prefix cross-reference targets, so that it is clear which intersphinx set the target belongs to. A link like `:ref:'comparison manual <python:comparisons>'` will link to the label "comparisons" in the doc set "python", if it exists.

Example

To add links to modules and objects in the Python standard library documentation, use:

```
intersphinx_mapping = {'python': ('http://docs.python.org/3.2', None)}
```

This will download the corresponding `objects.inv` file from the Internet and generate links to the pages under the given URI. The downloaded inventory is cached in the Sphinx environment, so it must be redownloaded whenever you do a full rebuild.

A second example, showing the meaning of a non-`None` value of the second tuple item:

```
intersphinx_mapping = {'python': ('http://docs.python.org/3.2',
                                  'python-inv.txt')}
```

This will read the inventory from `python-inv.txt` in the source directory, but still generate links to the pages under `http://docs.python.org/3.2`. It is up to you to update the inventory file as new objects are added to the Python documentation.

intersphinx_cache_limit

The maximum number of days to cache remote inventories. The default is `5`, meaning five days. Set this to a negative value to cache inventories for unlimited time.

11.4.5 Math support in Sphinx

New in version 0.5. Since mathematical notation isn't natively supported by HTML in any way, Sphinx supports math in documentation with two extensions.

The basic math support that is common to both extensions is contained in `sphinx.ext.mathbase`. Other math support extensions should, if possible, reuse that support too.

Note: `mathbase` is not meant to be added to the `extensions` config value, instead, use either `sphinx.ext.pngmath` or `sphinx.ext.jsmath` as described below.

The input language for mathematics is LaTeX markup. This is the de-facto standard for plain-text math notation and has the added advantage that no further translation is necessary when building LaTeX output.

`mathbase` defines these new markup elements:

:math:
Role for inline math. Use like this:

```
Since Pythagoras, we know that :math:`a^2 + b^2 = c^2`.
```

.. math::
Directive for displayed math (math that takes the whole line for itself).

The directive supports multiple equations, which should be separated by a blank line:

```
.. math::

   (a + b)^2 = a^2 + 2ab + b^2

   (a - b)^2 = a^2 - 2ab + b^2
```

In addition, each single equation is set within a `split` environment, which means that you can have multiple aligned lines in an equation, aligned at `&` and separated by `\\`:

```
.. math::

   (a + b)^2  &=  (a + b)(a + b) \\
              &=  a^2 + 2ab + b^2
```

For more details, look into the documentation of the AmSMath LaTeX package.

When the math is only one line of text, it can also be given as a directive argument:

```
.. math:: (a + b)^2 = a^2 + 2ab + b^2
```

Normally, equations are not numbered. If you want your equation to get a number, use the `label` option. When given, it selects a label for the equation, by which it can be cross-referenced, and causes an equation number to be issued. See `eqref` for an example. The numbering style depends on the output format.

There is also an option `nowrap` that prevents any wrapping of the given math in a math environment. When you give this option, you must make sure yourself that the math is properly set up. For example:

```
.. math::
   :nowrap:

   \begin{eqnarray}
      y    & = & ax^2 + bx + c \\
      f(x) & = & x^2 + 2xy + y^2
   \end{eqnarray}
```

:eq:
Role for cross-referencing equations via their label. This currently works only within the same document. Example:

```
.. math:: e^{i\pi} + 1 = 0
   :label: euler

Euler's identity, equation :eq:`euler`, was elected one of the most
beautiful mathematical formulas.
```

sphinx.ext.pngmath – Render math as PNG images

This extension renders math via LaTeX and dvipng into PNG images. This of course means that the computer where the docs are built must have both programs available.

There are various config values you can set to influence how the images are built:

pngmath_latex
The command name with which to invoke LaTeX. The default is `'latex'`; you may need to set this to a full path if `latex` is not in the executable search path.

Since this setting is not portable from system to system, it is normally not useful to set it in `conf.py`; rather, giving it on the **sphinx-build** command line via the `-D` option should be preferable, like this:

```
sphinx-build -b html -D pngmath_latex=C:\tex\latex.exe . _build/html
```

Changed in version 0.5.1: This value should only contain the path to the latex executable, not further arguments; use `pngmath_latex_args` for that purpose.

pngmath_dvipng
The command name with which to invoke `dvipng`. The default is `'dvipng'`; you may need to set this to a full path if `dvipng` is not in the executable search path.

pngmath_latex_args
Additional arguments to give to latex, as a list. The default is an empty list. New in version 0.5.1.

pngmath_latex_preamble
Additional LaTeX code to put into the preamble of the short LaTeX files that are used to translate the math snippets. This is empty by default. Use it e.g. to add more packages whose commands you want to use in the math.

pngmath_dvipng_args
Additional arguments to give to dvipng, as a list. The default value is `['-gamma 1.5', '-D 110']` which makes the image a bit darker and larger then it is by default.

An arguments you might want to add here is e.g. `'-bg Transparent'`, which produces PNGs with a transparent background. This is not enabled by default because some Internet Explorer versions don't like transparent PNGs.

Note: When you "add" an argument, you need to reproduce the default arguments if you want to keep them; that is, like this:

```
pngmath_dvipng_args = ['-gamma 1.5', '-D 110', '-bg Transparent']
```

---

pngmath_use_preview
`dvipng` has the ability to determine the "depth" of the rendered text: for example, when typesetting a fraction inline, the baseline of surrounding text should not be flush with the bottom of the image, rather the image should extend a bit below the baseline. This is what TeX calls "depth". When this is enabled, the images put into the HTML document will get a `vertical-align` style that correctly aligns the baselines.

Unfortunately, this only works when the preview-latex package is installed. Therefore, the default for this option is `False`.

sphinx.ext.jsmath – Render math via JavaScript

This extension puts math as-is into the HTML files. The JavaScript package jsMath is then loaded and transforms the LaTeX markup to readable math live in the browser.

Because jsMath (and the necessary fonts) is very large, it is not included in Sphinx. You must install it yourself, and give Sphinx its path in this config value:

jsmath_path
The path to the JavaScript file to include in the HTML files in order to load JSMath. There is no default.

The path can be absolute or relative; if it is relative, it is relative to the `_static` directory of the built docs.

For example, if you put JSMath into the static path of the Sphinx docs, this value would be `jsMath/easy/load.js`. If you host more than one Sphinx documentation set on one server, it is advisable to install jsMath in a shared location.

11.4.6 `sphinx.ext.graphviz` – Add Graphviz graphs

New in version 0.6. This extension allows you to embed Graphviz graphs in your documents.

It adds these directives:

.. graphviz::
Directive to embed graphviz code. The input code for `dot` is given as the content. For example:

```
.. graphviz::

   digraph foo {
      "bar" -> "baz";
   }
```

In HTML output, the code will be rendered to a PNG or SVG image (see `graphviz_output_format`). In LaTeX output, the code will be rendered to an embeddable PDF file.

.. graph::
Directive for embedding a single undirected graph. The name is given as a directive argument, the contents of the graph are the directive content. This is a convenience directive to generate `graph <name> { <content> }`.

For example:

```
.. graph:: foo

   "bar" -- "baz";
```

.. digraph::
Directive for embedding a single directed graph. The name is given as a directive argument, the contents of the graph are the directive content. This is a convenience directive to generate `digraph <name> { <content> }`.

For example:

```
.. digraph:: foo

   "bar" -> "baz" -> "quux";
```

New in version 1.0: All three directives support an `alt` option that determines the image's alternate text for HTML output. If not given, the alternate text defaults to the graphviz code. There are also these new config values:

graphviz_dot
The command name with which to invoke `dot`. The default is `'dot'`; you may need to set this to a full path if `dot` is not in the executable search path.

Since this setting is not portable from system to system, it is normally not useful to set it in `conf.py`; rather, giving it on the **sphinx-build** command line via the *-D* option should be preferable, like this:

```
sphinx-build -b html -D graphviz_dot=C:\graphviz\bin\dot.exe . _build/html
```

graphviz_dot_args
Additional command-line arguments to give to dot, as a list. The default is an empty list. This is the right place to set global graph, node or edge attributes via dot's `-G`, `-N` and `-E` options.

graphviz_output_format
The output format for Graphviz when building HTML files. This must be either `'png'` or `'svg'`; the default is `'png'`. New in version 1.0: Previously, output always was PNG.

11.4.7 `sphinx.ext.inheritance_diagram` – Include inheritance diagrams

New in version 0.6. This extension allows you to include inheritance diagrams, rendered via the Graphviz extension.

It adds this directive:

.. inheritance-diagram::
This directive has one or more arguments, each giving a module or class name. Class names can be unqualified; in that case they are taken to exist in the currently described module (see `py:module`).

For each given class, and each class in each given module, the base classes are determined. Then, from all classes and their base classes, a graph is generated which is then rendered via the graphviz extension to a directed graph.

This directive supports an option called `parts` that, if given, must be an integer, advising the directive to remove that many parts of module names from the displayed names. (For example, if all your class names start with `lib.`, you can give `:parts: 1` to remove that prefix from the displayed node names.)

New config values are:

inheritance_graph_attrs
A dictionary of graphviz graph attributes for inheritance diagrams.

For example:

```
inheritance_graph_attrs = dict(rankdir="LR", size='"6.0, 8.0"',
                               fontsize=14, ratio='compress')
```

inheritance_node_attrs
A dictionary of graphviz node attributes for inheritance diagrams.

For example:

```
inheritance_node_attrs = dict(shape='ellipse', fontsize=14, height=0.75,
                              color='dodgerblue1', style='filled')
```

inheritance_edge_attrs
A dictionary of graphviz edge attributes for inheritance diagrams.

11.4.8 sphinx.ext.refcounting – Keep track of reference counting behavior

Todo

Write this section.

11.4.9 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 registered 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.
```

To make a custom config value known to Sphinx, use `add_config_value()` in the setup function in `conf.py`, e.g.:

```
def setup(app):
    app.add_config_value('releaselevel', '', True)
```

The second argument is the default value, the third should always be `True` for such values (it selects if Sphinx re-reads the documents if the value changes).

11.4.10 sphinx.ext.coverage – Collect doc coverage stats

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

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

Todo

Write this section.

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

11.4.11 sphinx.ext.todo – Support for todo items

Module author: Daniel Bültmann New in version 0.5. There are two additional directives when using this extension:

.. todo::
Use this directive like, for example, note.

It will only show up in the output if todo_include_todos is true.

.. todolist::
This directive is replaced by a list of all todo directives in the whole documentation, if todo_include_todos is true.

There is also an additional config value:

todo_include_todos
If this is True, todo and todolist produce output, else they produce nothing. The default is False.

11.4.12 sphinx.ext.extlinks – Markup to shorten external links

Module author: Georg Brandl New in version 1.0. This extension is meant to help with the common pattern of having many external links that point to URLs on one and the same site, e.g. links to bug trackers, version control web interfaces, or simply subpages in other websites. It does so by providing aliases to base URLs, so that you only need to give the subpage name when creating a link.

Let's assume that you want to include many links to issues at the Sphinx tracker, at 'http://bitbucket.org/birkenfeld/sphinx/issue/num'. Typing this URL again and again is tedious, so you can use extlinks to avoid repeating yourself.

The extension adds one new config value:

extlinks
This config value must be a dictionary of external sites, mapping unique short alias names to a base URL and a *prefix*. For example, to create an alias for the above mentioned issues, you would add

```
extlinks = {'issue': ('http://bitbucket.org/birkenfeld/sphinx/issue/%s',
                      'issue ')}
```

Now, you can use the alias name as a new role, e.g. :issue:`123`. This then inserts a link to http://bitbucket.org/birkenfeld/sphinx/issue/123. As you can see, the target given in the role is substituted in the base URL in the place of %s.

The link *caption* depends on the second item in the tuple, the *prefix*:

- If the prefix is None, the link caption is the full URL.
- If the prefix is the empty string, the link caption is the partial URL given in the role content (123 in this case.)
- If the prefix is a non-empty string, the link caption is the partial URL, prepended by the prefix – in the above example, the link caption would be issue 123.

You can also use the usual "explicit title" syntax supported by other roles that generate links, i.e. :issue:`this issue <123>`. In this case, the *prefix* is not relevant.

Note: Since links are generated from the role in the reading stage, they appear as ordinary links to e.g. the linkcheck builder.

11.4.13 sphinx.ext.viewcode – Add links to highlighted source code

Module author: Georg Brandl New in version 1.0. This extension looks at your Python object descriptions (`.. class::`, `.. function::` etc.) and tries to find the source files where the objects are contained. When found, a separate HTML page will be output for each module with a highlighted version of the source code, and a link will be added to all object descriptions that leads to the source code of the described object. A link back from the source to the description will also be inserted.

There are currently no configuration values for this extension; you just need to add `'sphinx.ext.viewcode'` to your `extensions` value for it to work.

11.4.14 sphinx.ext.oldcmarkup – Compatibility extension for old C markup

Module author: Georg Brandl New in version 1.0. This extension is a transition helper for projects that used the old (pre-domain) C markup, i.e. the directives like `cfunction` and roles like `cfunc`. Since the introduction of domains, they must be called by their fully-qualified name (`c:function` and `c:func`, respectively) or, with the default domain set to `c`, by their new name (`function` and `func`). (See *The C Domain* for the details.)

If you activate this extension, it will register the old names, and you can use them like before Sphinx 1.0. The directives are:

- `cfunction`
- `cmember`
- `cmacro`
- `ctype`
- `cvar`

The roles are:

- `cdata`
- `cfunc`
- `cmacro`
- `ctype`

However, it is advised to migrate to the new markup – this extension is a compatibility convenience and will disappear in a future version of Sphinx.

11.5 Third-party extensions

You can find several extensions contributed by users in the Sphinx Contrib repository. It is open for anyone who wants to maintain an extension publicly; just send a short message asking for write permissions.

There are also several extensions hosted elsewhere. The Wiki at BitBucket maintains a list of those.

If you write an extension that you think others will find useful or you think should be included as a part of Sphinx, please write to the project mailing list (join here).

11.5.1 Where to put your own extensions?

Extensions local to a project should be put within the project's directory structure. Set Python's module search path, `sys.path`, accordingly so that Sphinx can find them. E.g., if your extension `foo.py` lies in the `exts` subdirectory of the project root, put into `conf.py`:

```
import sys, os

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

extensions = ['foo']
```

You can also install extensions anywhere else on `sys.path`, e.g. in the `site-packages` directory.

CHAPTER

TWELVE

SPHINX FAQ

This is a list of Frequently Asked Questions about Sphinx. Feel free to suggest new entries!

12.1 How do I...

... create PDF files without LaTeX? You can use rst2pdf version 0.12 or greater which comes with built-in Sphinx integration. See the *Available builders* section for details.

... get section numbers? They are automatic in LaTeX output; for HTML, give a `:numbered:` option to the `toctree` directive where you want to start numbering.

... customize the look of the built HTML files? Use themes, see *HTML theming support*.

... add global substitutions or includes? Add them in the `rst_epilog` config value.

... display the whole TOC tree in the sidebar? Use the `toctree` callable in a custom layout template, probably in the `sidebartoc` block.

... write my own extension? See the *extension tutorial*.

... convert from my existing docs using MoinMoin markup? The easiest way is to convert to xhtml, then convert xhtml to reST. You'll still need to mark up classes and such, but the headings and code examples come through cleanly.

12.2 Using Sphinx with...

Epydoc There's a third-party extension providing an api role which refers to Epydoc's API docs for a given identifier.

Doxygen Michael Jones is developing a reST/Sphinx bridge to doxygen called breathe.

SCons Glenn Hutchings has written a SCons build script to build Sphinx documentation; it is hosted here: http://bitbucket.org/zondo/sphinx-scons

PyPI Jannis Leidel wrote a setuptools command that automatically uploads Sphinx documentation to the PyPI package documentation area at http://packages.python.org/.

MediaWiki See http://bitbucket.org/kevindunn/sphinx-wiki, a project by Kevin Dunn.

github pages You'll have to opt out of processing your pages with the "Jekyll" preprocessor as described in http://pages.github.com/#using_jekyll_for_complex_layouts.

Google Analytics You can use a custom `layout.html` template, like this:

```
{% extends "!layout.html" %}

{%- block extrahead %}
{{ super() }}
<script type="text/javascript">
  var _gaq = _gaq || [];
  _gaq.push(['_setAccount', 'XXX account number XXX']);
  _gaq.push(['_trackPageview']);
</script>
{% endblock %}

{% block footer %}
{{ super() }}
<div class="footer">This page uses <a href="http://analytics.google.com/">
Google Analytics</a> to collect statistics. You can disable it by blocking
the JavaScript coming from www.google-analytics.com.
<script type="text/javascript">
  (function() {
    var ga = document.createElement('script');
    ga.src = ('https:' == document.location.protocol ?
              'https://ssl' : 'http://www') + '.google-analytics.com/ga.js';
    ga.setAttribute('async', 'true');
    document.documentElement.firstChild.appendChild(ga);
  })();
</script>
</div>
{% endblock %}
```

12.3 Epub info

The epub builder is currently in an experimental stage. It has only been tested with the Sphinx documentation itself. If you want to create epubs, here are some notes:

- Split the text into several files. The longer the individual HTML files are, the longer it takes the ebook reader to render them. In extreme cases, the rendering can take up to one minute.
- Try to minimize the markup. This also pays in rendering time.
- For some readers you can use embedded or external fonts using the CSS `@font-face` directive. This is *extremely* useful for code listings which are often cut at the right margin. The default Courier font (or variant) is quite wide and you can only display up to 60 characters on a line. If you replace it with a narrower font, you can get more characters on a line. You may even use FontForge and create narrow variants of some free font. In my case I get up to 70 characters on a line.

  You may have to experiment a little until you get reasonable results.
- Test the created epubs. You can use several alternatives. The ones I am aware of are Epubcheck, Calibre, FBreader (although it does not render the CSS), and Bookworm. For bookworm you can download the source from http://code.google.com/p/threepress/ and run your own local server.
- Large floating divs are not displayed properly. If they cover more than one page, the div is only shown on the first page. In that case you can copy the `epub.css` from the `sphinx/themes/epub/static/` directory to your local `_static/` directory and remove the float settings.
- Files that are inserted outside of the `toctree` directive must be manually included. This sometimes applies to appendixes, e.g. the glossary or the indices. You can add them with the `epub_post_files`

option.

CHAPTER

THIRTEEN

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.

directive A reStructuredText markup element that allows marking a block of content with special meaning. Directives are supplied not only by docutils, but Sphinx and custom extensions can add their own. The basic directive syntax looks like this:

```
.. directivename:: argument ...
   :option: value

   Content of the directive.
```

See *Directives* for more information.

document name Since 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: *document names* are always relative to the *source directory*, the extension is stripped, and path separators are converted to slashes. All values, parameters and such referring to "documents" expect such document names.

Examples for document names are `index`, `library/zipfile`, or `reference/datamodel/types`. Note that there is no leading or trailing slash.

domain A domain is a collection of markup (reStructuredText *directive*s and *role*s) to describe and link to *object*s belonging together, e.g. elements of a programming language. Directive and role names in a domain have names like `domain:name`, e.g. `py:function`.

Having domains means that there are no naming problems when one set of documentation wants to refer to e.g. C++ and Python classes. It also means that extensions that support the documentation of whole new languages are much easier to write. For more information about domains, see the chapter *Sphinx Domains*.

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.

master document The document that contains the root `toctree` directive.

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

role A reStructuredText markup element that allows marking a piece of text. Like directives, roles are extensible. The basic syntax looks like this: `:rolename:`content``. See *Inline markup* for details.

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

CHAPTER

FOURTEEN

CHANGES IN SPHINX

14.1 Release 1.0.8 (Sep 23, 2011)

- #627: Fix tracebacks for AttributeErrors in autosummary generation.
- Fix the `abbr` role when the abbreviation has newlines in it.
- #727: Fix the links to search results with custom object types.
- #648: Fix line numbers reported in warnings about undefined references.
- #696, #666: Fix C++ array definitions and template arguments that are not type names.
- #633: Allow footnotes in section headers in LaTeX output.
- #616: Allow keywords to be linked via intersphinx.
- #613: Allow Unicode characters in production list token names.
- #720: Add dummy visitors for graphviz nodes for text and man.
- #704: Fix image file duplication bug.
- #677: Fix parsing of multiple signatures in C++ domain.
- #637: Ignore Emacs lock files when looking for source files.
- #544: Allow .pyw extension for importable modules in autodoc.
- #700: Use `$(MAKE)` in quickstart-generated Makefiles.
- #734: Make sidebar search box width consistent in browsers.
- #644: Fix spacing of centered figures in HTML output.
- #767: Safely encode SphinxError messages when printing them to sys.stderr.
- #611: Fix LaTeX output error with a document with no sections but a link target.
- Correctly treat built-in method descriptors as methods in autodoc.
- #706: Stop monkeypatching the Python textwrap module.
- #657: viewcode now works correctly with source files that have non-ASCII encoding.
- #669: Respect the `noindex` flag option in py:module directives.
- #675: Fix IndexErrors when including nonexisting lines with `literalinclude`.
- #676: Respect custom function/method parameter separator strings.
- #682: Fix JS incompatibility with jQuery >= 1.5.

- #693: Fix double encoding done when writing HTMLHelp .hhk files.
- #647: Do not apply SmartyPants in parsed-literal blocks.
- C++ domain now supports array definitions.

14.2 Release 1.0.7 (Jan 15, 2011)

- #347: Fix wrong generation of directives of static methods in autosummary.
- #599: Import PIL as `from PIL import Image`.
- #558: Fix longtables with captions in LaTeX output.
- Make token references work as hyperlinks again in LaTeX output.
- #572: Show warnings by default when reference labels cannot be found.
- #536: Include line number when complaining about missing reference targets in nitpicky mode.
- #590: Fix inline display of graphviz diagrams in LaTeX output.
- #589: Build using app.build() in setup command.
- Fix a bug in the inheritance diagram exception that caused base classes to be skipped if one of them is a builtin.
- Fix general index links for C++ domain objects.
- #332: Make admonition boundaries in LaTeX output visible.
- #573: Fix KeyErrors occurring on rebuild after removing a file.
- Fix a traceback when removing files with globbed toctrees.
- If an autodoc object cannot be imported, always re-read the document containing the directive on next build.
- If an autodoc object cannot be imported, show the full traceback of the import error.
- Fix a bug where the removal of download files and images wasn't noticed.
- #571: Implement ~ cross-reference prefix for the C domain.
- Fix regression of LaTeX output with the fix of #556.
- #568: Fix lookup of class attribute documentation on descriptors so that comment documentation now works.
- Fix traceback with `only` directives preceded by targets.
- Fix tracebacks occurring for duplicate C++ domain objects.
- Fix JavaScript domain links to objects with `$` in their name.

14.3 Release 1.0.6 (Jan 04, 2011)

- #581: Fix traceback in Python domain for empty cross-reference targets.
- #283: Fix literal block display issues on Chrome browsers.
- #383, #148: Support sorting a limited range of accented characters in the general index and the glossary.

- #570: Try decoding `-D` and `-A` command-line arguments with the locale's preferred encoding.
- #528: Observe `locale_dirs` when looking for the JS translations file.
- #574: Add special code for better support of Japanese documents in the LaTeX builder.
- Regression of #77: If there is only one parameter given with `:param:` markup, the bullet list is now suppressed again.
- #556: Fix missing paragraph breaks in LaTeX output in certain situations.
- #567: Emit the `autodoc-process-docstring` event even for objects without a docstring so that it can add content.
- #565: In the LaTeX builder, not only literal blocks require different table handling, but also quite a few other list-like block elements.
- #515: Fix tracebacks in the viewcode extension for Python objects that do not have a valid signature.
- Fix strange reportings of line numbers for warnings generated from autodoc-included docstrings, due to different behavior depending on docutils version.
- Several fixes to the C++ domain.

14.4 Release 1.0.5 (Nov 12, 2010)

- #557: Add CSS styles required by docutils 0.7 for aligned images and figures.
- In the Makefile generated by LaTeX output, do not delete pdf files on clean; they might be required images.
- #535: Fix LaTeX output generated for line blocks.
- #544: Allow `.pyw` as a source file extension.

14.5 Release 1.0.4 (Sep 17, 2010)

- #524: Open intersphinx inventories in binary mode on Windows, since version 2 contains zlib-compressed data.
- #513: Allow giving non-local URIs for JavaScript files, e.g. in the JSMath extension.
- #512: Fix traceback when `intersphinx_mapping` is empty.

14.6 Release 1.0.3 (Aug 23, 2010)

- #495: Fix internal vs. external link distinction for links coming from a docutils table-of-contents.
- #494: Fix the `maxdepth` option for the `toctree()` template callable when used with `collapse=True`.
- #507: Fix crash parsing Python argument lists containing brackets in string literals.
- #501: Fix regression when building LaTeX docs with figures that don't have captions.
- #510: Fix inheritance diagrams for classes that are not picklable.
- #497: Introduce separate background color for the sidebar collapse button, making it easier to see.

- #502, #503, #496: Fix small layout bugs in several builtin themes.

14.7 Release 1.0.2 (Aug 14, 2010)

- #490: Fix cross-references to objects of types added by the `add_object_type()` API function.
- Fix handling of doc field types for different directive types.
- Allow breaking long signatures, continuing with backlash-escaped newlines.
- Fix unwanted styling of C domain references (because of a namespace clash with Pygments styles).
- Allow references to PEPs and RFCs with explicit anchors.
- #471: Fix LaTeX references to figures.
- #482: When doing a non-exact search, match only the given type of object.
- #481: Apply non-exact search for Python reference targets with `.name` for modules too.
- #484: Fix crash when duplicating a parameter in an info field list.
- #487: Fix setting the default role to one provided by the `oldcmarkup` extension.
- #488: Fix crash when json-py is installed, which provides a `json` module but is incompatible to simplejson.
- #480: Fix handling of target naming in intersphinx.
- #486: Fix removal of `!` for all cross-reference roles.

14.8 Release 1.0.1 (Jul 27, 2010)

- #470: Fix generated target names for reST domain objects; they are not in the same namespace.
- #266: Add Bengali language.
- #473: Fix a bug in parsing JavaScript object names.
- #474: Fix building with SingleHTMLBuilder when there is no toctree.
- Fix display names for objects linked to by intersphinx with explicit targets.
- Fix building with the JSON builder.
- Fix hyperrefs in object descriptions for LaTeX.

14.9 Release 1.0 (Jul 23, 2010)

14.9.1 Incompatible changes

- Support for domains has been added. A domain is a collection of directives and roles that all describe objects belonging together, e.g. elements of a programming language. A few builtin domains are provided:
  - Python
  - C

  - C++
  - JavaScript
  - reStructuredText
- The old markup for defining and linking to C directives is now deprecated. It will not work anymore in future versions without activating the `oldcmarkup` extension; in Sphinx 1.0, it is activated by default.
- Removed support for old dependency versions; requirements are now:
  - docutils >= 0.5
  - Jinja2 >= 2.2
- Removed deprecated elements:
  - `exclude_dirs` config value
  - `sphinx.builder` module

14.9.2 Features added

- General:
  - Added a "nitpicky" mode that emits warnings for all missing references. It is activated by the `-n` command-line switch or the `nitpicky` config value.
  - Added `latexpdf` target in quickstart Makefile.
- Markup:
  - The `menuselection` and `guilabel` roles now support ampersand accelerators.
  - New more compact doc field syntax is now recognized: `:param type name: description`.
  - Added `tab-width` option to `literalinclude` directive.
  - Added `titlesonly` option to `toctree` directive.
  - Added the `prepend` and `append` options to the `literalinclude` directive.
  - #284: All docinfo metadata is now put into the document metadata, not just the author.
  - The `ref` role can now also reference tables by caption.
  - The `include` directive now supports absolute paths, which are interpreted as relative to the source directory.
  - In the Python domain, references like `:func:`.name`` now look for matching names with any prefix if no direct match is found.
- Configuration:
  - Added `rst_prolog` config value.
  - Added `html_secnumber_suffix` config value to control section numbering format.
  - Added `html_compact_lists` config value to control docutils' compact lists feature.
  - The `html_sidebars` config value can now contain patterns as keys, and the values can be lists that explicitly select which sidebar templates should be rendered. That means that the builtin sidebar contents can be included only selectively.
  - `html_static_path` can now contain single file entries.

  - The new universal config value `exclude_patterns` makes the old `unused_docs`, `exclude_trees` and `exclude_dirnames` obsolete.
  - Added `html_output_encoding` config value.
  - Added the `latex_docclass` config value and made the "twoside" documentclass option overridable by "oneside".
  - Added the `trim_doctest_flags` config value, which is true by default.
  - Added `html_show_copyright` config value.
  - Added `latex_show_pagerefs` and `latex_show_urls` config values.
  - The behavior of `html_file_suffix` changed slightly: the empty string now means "no suffix" instead of "default suffix", use `None` for "default suffix".
- New builders:
  - Added a builder for the Epub format.
  - Added a builder for manual pages.
  - Added a single-file HTML builder.
- HTML output:
  - Inline roles now get a CSS class with their name, allowing styles to customize their appearance. Domain-specific roles get two classes, `domain` and `domain-rolename`.
  - References now get the class `internal` if they are internal to the whole project, as opposed to internal to the current page.
  - External references can be styled differently with the new `externalrefs` theme option for the default theme.
  - In the default theme, the sidebar can experimentally now be made collapsible using the new `collapsiblesidebar` theme option.
  - #129: Toctrees are now wrapped in a `div` tag with class `toctree-wrapper` in HTML output.
  - The `toctree` callable in templates now has a `maxdepth` keyword argument to control the depth of the generated tree.
  - The `toctree` callable in templates now accepts a `titles_only` keyword argument.
  - Added `htmltitle` block in layout template.
  - In the JavaScript search, allow searching for object names including the module name, like `sys.argv`.
  - Added new theme `haiku`, inspired by the Haiku OS user guide.
  - Added new theme `nature`.
  - Added new theme `agogo`, created by Andi Albrecht.
  - Added new theme `scrolls`, created by Armin Ronacher.
  - #193: Added a `visitedlinkcolor` theme option to the default theme.
  - #322: Improved responsiveness of the search page by loading the search index asynchronously.
- Extension API:
  - Added `html-collect-pages`.
  - Added `needs_sphinx` config value and `require_sphinx()` application API method.

  - #200: Added `add_stylesheet()` application API method.

- Extensions:
  - Added the `viewcode` extension.
  - Added the `extlinks` extension.
  - Added support for source ordering of members in autodoc, with `autodoc_member_order = 'bysource'`.
  - Added `autodoc_default_flags` config value, which can be used to select default flags for all autodoc directives.
  - Added a way for intersphinx to refer to named labels in other projects, and to specify the project you want to link to.
  - #280: Autodoc can now document instance attributes assigned in `__init__` methods.
  - Many improvements and fixes to the `autosummary` extension, thanks to Pauli Virtanen.
  - #309: The `graphviz` extension can now output SVG instead of PNG images, controlled by the `graphviz_output_format` config value.
  - Added `alt` option to `graphviz` extension directives.
  - Added `exclude` argument to `autodoc.between()`.
- Translations:
  - Added Croatian translation, thanks to Bojan Mihelač.
  - Added Turkish translation, thanks to Firat Ozgul.
  - Added Catalan translation, thanks to Pau Fernández.
  - Added simplified Chinese translation.
  - Added Danish translation, thanks to Hjorth Larsen.
  - Added Lithuanian translation, thanks to Dalius Dobravolskas.
- Bugs fixed:
  - #445: Fix links to result pages when using the search function of HTML built with the `dirhtml` builder.
  - #444: In templates, properly re-escape values treated with the "striptags" Jinja filter.

14.10 Release 0.6.7 (Jun 05, 2010)

- #440: Remove usage of a Python >= 2.5 API in the `literalinclude` directive.
- Fix a bug that prevented some references being generated in the LaTeX builder.
- #428: Add some missing CSS styles for standard docutils classes.
- #432: Fix UnicodeErrors while building LaTeX in translated locale.

14.11 Release 0.6.6 (May 25, 2010)

- Handle raw nodes in the `text` writer.
- Fix a problem the Qt help project generated by the `qthelp` builder that would lead to no content being displayed in the Qt Assistant.
- #393: Fix the usage of Unicode characters in mathematic formulas when using the `pngmath` extension.
- #404: Make `\and` work properly in the author field of the `latex_documents` setting.
- #409: Make the `highlight_language` config value work properly in the LaTeX builder.
- #418: Allow relocation of the translation JavaScript files to the system directory on Unix systems.
- #414: Fix handling of Windows newlines in files included with the `literalinclude` directive.
- #377: Fix crash in linkcheck builder.
- #387: Fix the display of search results in `dirhtml` output.
- #376: In autodoc, fix display of parameter defaults containing backslashes.
- #370: Fix handling of complex list item labels in LaTeX output.
- #374: Make the `doctest_path` config value of the doctest extension actually work.
- Fix the handling of multiple toctrees when creating the global TOC for the `toctree()` template function.
- Fix the handling of hidden toctrees when creating the global TOC for the `toctree()` template function.
- Fix the handling of nested lists in the text writer.
- #362: In autodoc, check for the existence of `__self__` on function objects before accessing it.
- #353: Strip leading and trailing whitespace when extracting search words in the search function.

14.12 Release 0.6.5 (Mar 01, 2010)

- In autodoc, fix the omission of some module members explicitly documented using documentation comments.
- #345: Fix cropping of sidebar scroll bar with `stickysidebar` option of the default theme.
- #341: Always generate UNIX newlines in the quickstart Makefile.
- #338: Fix running with `-C` under Windows.
- In autodoc, allow customizing the signature of an object where the built-in mechanism fails.
- #331: Fix output for enumerated lists with start values in LaTeX.
- Make the `start-after` and `end-before` options to the `literalinclude` directive work correctly if not used together.
- #321: Fix link generation in the LaTeX builder.

14.13 Release 0.6.4 (Jan 12, 2010)

- Improve the handling of non-Unicode strings in the configuration.
- #316: Catch OSErrors occurring when calling graphviz with arguments it doesn't understand.
- Restore compatibility with Pygments >= 1.2.
- #295: Fix escaping of hyperref targets in LaTeX output.
- #302: Fix links generated by the `:doc:` role for LaTeX output.
- #286: collect todo nodes after the whole document has been read; this allows placing substitution references in todo items.
- #294: do not ignore an explicit `today` config value in a LaTeX build.
- The `alt` text of inheritance diagrams is now much cleaner.
- Ignore images in section titles when generating link captions.
- #310: support exception messages in the `testoutput` blocks of the `doctest` extension.
- #293: line blocks are styled properly in HTML output.
- #285: make the `locale_dirs` config value work again.
- #303: `html_context` values given on the command line via `-A` should not override other values given in conf.py.
- Fix a bug preventing incremental rebuilds for the `dirhtml` builder.
- #299: Fix the mangling of quotes in some literal blocks.
- #292: Fix path to the search index for the `dirhtml` builder.
- Fix a Jython compatibility issue: make the dependence on the `parser` module optional.
- #238: In autodoc, catch all errors that occur on module import, not just `ImportError`.
- Fix the handling of non-data, but non-method descriptors in autodoc.
- When copying file times, ignore OSErrors raised by `os.utime()`.

14.14 Release 0.6.3 (Sep 03, 2009)

- Properly add C module filenames as dependencies in autodoc.
- #253: Ignore graphviz directives without content instead of raising an unhandled exception.
- #241: Fix a crash building LaTeX output for documents that contain a todolist directive.
- #252: Make it easier to change the build dir in the Makefiles generated by quickstart.
- #220: Fix CSS so that displaymath really is centered.
- #222: Allow the "Footnotes" header to be translated.
- #225: Don't add whitespace in generated HTML after inline tags.
- #227: Make `literalinclude` work when the document's path name contains non-ASCII characters.
- #229: Fix autodoc failures with members that raise errors on `getattr()`.
- #205: When copying files, don't copy full stat info, only modification times.

- #232: Support non-ASCII metadata in Qt help builder.
- Properly format bullet lists nested in definition lists for LaTeX.
- Section titles are now allowed inside `only` directives.
- #201: Make `centered` directive work in LaTeX output.
- #206: Refuse to overwrite an existing master document in sphinx-quickstart.
- #208: Use MS-sanctioned locale settings, determined by the `language` config option, in the HTML help builder.
- #210: Fix nesting of HTML tags for displayed math from pngmath extension.
- #213: Fix centering of images in LaTeX output.
- #211: Fix compatibility with docutils 0.5.

14.15 Release 0.6.2 (Jun 16, 2009)

- #130: Fix obscure IndexError in doctest extension.
- #167: Make glossary sorting case-independent.
- #196: Add a warning if an extension module doesn't have a `setup()` function.
- #158: Allow '..' in template names, and absolute template paths; Jinja 2 by default disables both.
- When highlighting Python code, ignore extra indentation before trying to parse it as Python.
- #191: Don't escape the tilde in URIs in LaTeX.
- Don't consider contents of source comments for the search index.
- Set the default encoding to `utf-8-sig` to handle files with a UTF-8 BOM correctly.
- #178: apply `add_function_parentheses` config value to C functions as promised.
- #173: Respect the docutils `title` directive.
- #172: The `obj` role now links to modules as promised.
- #19: Tables now can have a "longtable" class, in order to get correctly broken into pages in LaTeX output.
- Look for Sphinx message catalogs in the system default path before trying `sphinx/locale`.
- Fix the search for methods via "classname.methodname".
- #155: Fix Python 2.4 compatibility: exceptions are old-style classes there.
- #150: Fix display of the "sphinxdoc" theme on Internet Explorer versions 6 and 7.
- #146: Don't fail to generate LaTeX when the user has an active `.docutils` configuration.
- #29: Don't generate visible "-{-}" in option lists in LaTeX.
- Fix cross-reference roles when put into substitutions.
- Don't put image "alt" text into table-of-contents entries.
- In the LaTeX writer, do not raise an exception on too many section levels, just use the "subparagraph" level for all of them.
- #145: Fix autodoc problem with automatic members that refuse to be getattr()'d from their parent.

- If specific filenames to build are given on the command line, check that they are within the source directory.
- Fix autodoc crash for objects without a `__name__`.
- Fix intersphinx for installations without urllib2.HTTPSHandler.
- #134: Fix pending_xref leftover nodes when using the todolist directive from the todo extension.

14.16 Release 0.6.1 (Mar 26, 2009)

- #135: Fix problems with LaTeX output and the graphviz extension.
- #132: Include the autosummary "module" template in the distribution.

14.17 Release 0.6 (Mar 24, 2009)

14.17.1 New features added

- Incompatible changes:
  - Templating now requires the Jinja2 library, which is an enhanced version of the old Jinja1 engine. Since the syntax and semantic is largely the same, very few fixes should be necessary in custom templates.
  - The "document" div tag has been moved out of the `layout.html` template's "document" block, because the closing tag was already outside. If you overwrite this block, you need to remove your "document" div tag as well.
  - The `autodoc_skip_member` event now also gets to decide whether to skip members whose name starts with underscores. Previously, these members were always automatically skipped. Therefore, if you handle this event, add something like this to your event handler to restore the old behavior:

    ```
    if name.startswith('_'):
        return True
    ```

- Theming support, see the new section in the documentation.
- Markup:
  - Due to popular demand, added a `:doc:` role which directly links to another document without the need of creating a label to which a `:ref:` could link to.
  - #4: Added a `:download:` role that marks a non-document file for inclusion into the HTML output and links to it.
  - Added an `only` directive that can selectively include text based on enabled "tags". Tags can be given on the command line. Also, the current builder output format (e.g. "html" or "latex") is always a defined tag.
  - #10: Added HTML section numbers, enabled by giving a `:numbered:` flag to the `toctree` directive.
  - #114: Added an `abbr` role to markup abbreviations and acronyms.
  - The `literalinclude` directive now supports several more options, to include only parts of a file.

  - The `toctree` directive now supports a `:hidden:` flag, which will prevent links from being generated in place of the directive – this allows you to define your document structure, but place the links yourself.
  - #123: The `glossary` directive now supports a `:sorted:` flag that sorts glossary entries alphabetically.
  - Paths to images, literal include files and download files can now be absolute (like `/images/foo.png`). They are treated as relative to the top source directory.
  - #52: There is now a `hlist` directive, creating a compact list by placing distributing items into multiple columns.
  - #77: If a description environment with info field list only contains one `:param:` entry, no bullet list is generated.
  - #6: Don't generate redundant `<ul>` for top-level TOC tree items, which leads to a visual separation of TOC entries.
  - #23: Added a `classmethod` directive along with `method` and `staticmethod`.
  - Scaled images now get a link to the unscaled version.
  - SVG images are now supported in HTML (via `<object>` and `<embed>` tags).
  - Added a `toctree` callable to the templates, and the ability to include external links in toctrees. The 'collapse' keyword argument indicates whether or not to only display subitems of the current page. (Defaults to True.)

- Configuration:
  - The new config value `rst_epilog` can contain reST that is appended to each source file that is read. This is the right place for global substitutions.
  - The new `html_add_permalinks` config value can be used to switch off the generated "paragraph sign" permalinks for each heading and definition environment.
  - The new `html_show_sourcelink` config value can be used to switch off the links to the reST sources in the sidebar.
  - The default value for `htmlhelp_basename` is now the project title, cleaned up as a filename.
  - The new `modindex_common_prefix` config value can be used to ignore certain package names for module index sorting.
  - The new `trim_footnote_reference_space` config value mirrors the docutils config value of the same name and removes the space before a footnote reference that is necessary for reST to recognize the reference.
  - The new `latex_additional_files` config value can be used to copy files (that Sphinx doesn't copy automatically, e.g. if they are referenced in custom LaTeX added in `latex_elements`) to the build directory.

- Builders:
  - The HTML builder now stores a small file named `.buildinfo` in its output directory. It stores a hash of config values that can be used to determine if a full rebuild needs to be done (e.g. after changing `html_theme`).
  - New builder for Qt help collections, by Antonio Valentino.
  - The new `DirectoryHTMLBuilder` (short name `dirhtml`) creates a separate directory for every page, and places the page there in a file called `index.html`. Therefore, page URLs and links don't need to contain `.html`.

  - The new `html_link_suffix` config value can be used to select the suffix of generated links between HTML files.
  - #96: The LaTeX builder now supports figures wrapped by text, when using the `figwidth` option and right/left alignment.
- New translations:
  - Italian by Sandro Dentella.
  - Ukrainian by Petro Sasnyk.
  - Finnish by Jukka Inkeri.
  - Russian by Alexander Smishlajev.
- Extensions and API:
  - New `graphviz` extension to embed graphviz graphs.
  - New `inheritance_diagram` extension to embed... inheritance diagrams!
  - New `autosummary` extension that generates summaries of modules and automatic documentation of modules.
  - Autodoc now has a reusable Python API, which can be used to create custom types of objects to auto-document (e.g. Zope interfaces). See also `Sphinx.add_autodocumenter()`.
  - Autodoc now handles documented attributes.
  - Autodoc now handles inner classes and their methods.
  - Autodoc can document classes as functions now if explicitly marked with *autofunction*.
  - Autodoc can now exclude single members from documentation via the `exclude-members` option.
  - Autodoc can now order members either alphabetically (like previously) or by member type; configurable either with the config value `autodoc_member_order` or a `member-order` option per directive.
  - The function `Sphinx.add_directive()` now also supports docutils 0.5-style directive classes. If they inherit from `sphinx.util.compat.Directive`, they also work with docutils 0.4.
  - There is now a `Sphinx.add_lexer()` method to be able to use custom Pygments lexers easily.
  - There is now `Sphinx.add_generic_role()` to mirror the docutils' own function.
- Other changes:
  - Config overrides for single dict keys can now be given on the command line.
  - There is now a `doctest_global_setup` config value that can be used to give setup code for all doctests in the documentation.
  - Source links in HTML are now generated with `rel="nofollow"`.
  - Quickstart can now generate a Windows `make.bat` file.
  - #62: There is now a `-w` option for sphinx-build that writes warnings to a file, in addition to stderr.
  - There is now a `-W` option for sphinx-build that turns warnings into errors.

14.18 Release 0.5.2 (Mar 24, 2009)

- Properly escape | in LaTeX output.
- #71: If a decoding error occurs in source files, print a warning and replace the characters by "?".
- Fix a problem in the HTML search if the index takes too long to load.
- Don't output system messages while resolving, because they would stay in the doctrees even if keep_warnings is false.
- #82: Determine the correct path for dependencies noted by docutils. This fixes behavior where a source with dependent files was always reported as changed.
- Recognize toctree directives that are not on section toplevel, but within block items, such as tables.
- Use a new RFC base URL, since rfc.org seems down.
- Fix a crash in the todolist directive when no todo items are defined.
- Don't call LaTeX or dvipng over and over again if it was not found once, and use text-only latex as a substitute in that case.
- Fix problems with footnotes in the LaTeX output.
- Prevent double hyphens becoming en-dashes in literal code in the LaTeX output.
- Open literalinclude files in universal newline mode to allow arbitrary newline conventions.
- Actually make the `-Q` option work.
- #86: Fix explicit document titles in toctrees.
- #81: Write environment and search index in a manner that is safe from exceptions that occur during dumping.
- #80: Fix UnicodeErrors when a locale is set with setlocale().

14.19 Release 0.5.1 (Dec 15, 2008)

- #67: Output warnings about failed doctests in the doctest extension even when running in quiet mode.
- #72: In pngmath, make it possible to give a full path to LaTeX and dvipng on Windows. For that to work, the `pngmath_latex` and `pngmath_dvipng` options are no longer split into command and additional arguments; use `pngmath_latex_args` and `pngmath_dvipng_args` to give additional arguments.
- Don't crash on failing doctests with non-ASCII characters.
- Don't crash on writing status messages and warnings containing unencodable characters.
- Warn if a doctest extension block doesn't contain any code.
- Fix the handling of `:param:` and `:type:` doc fields when they contain markup (especially cross-referencing roles).
- #65: Fix storage of depth information for PNGs generated by the pngmath extension.
- Fix autodoc crash when automethod is used outside a class context.
- #68: Fix LaTeX writer output for images with specified height.
- #60: Fix wrong generated image path when including images in sources in subdirectories.

- Fix the JavaScript search when html_copy_source is off.
- Fix an indentation problem in autodoc when documenting classes with the option `autoclass_content = "both"` set.
- Don't crash on empty index entries, only emit a warning.
- Fix a typo in the search JavaScript code, leading to unusable search function in some setups.

14.20 Release 0.5 (Nov 23, 2008) – Birthday release!

14.20.1 New features added

- Markup features:
  - Citations are now global: all citation defined in any file can be referenced from any file. Citations are collected in a bibliography for LaTeX output.
  - Footnotes are now properly handled in the LaTeX builder: they appear at the location of the footnote reference in text, not at the end of a section. Thanks to Andrew McNamara for the initial patch.
  - "System Message" warnings are now automatically removed from the built documentation, and only written to stderr. If you want the old behavior, set the new config value `keep_warnings` to True.
  - Glossary entries are now automatically added to the index.
  - Figures with captions can now be referred to like section titles, using the `:ref:` role without an explicit link text.
  - Added `cmember` role for consistency.
  - Lists enumerated by letters or roman numerals are now handled like in standard reST.
  - The `seealso` directive can now also be given arguments, as a short form.
  - You can now document several programs and their options with the new `program` directive.
- HTML output and templates:
  - Incompatible change: The "root" relation link (top left in the relbar) now points to the `master_doc` by default, no longer to a document called "index". The old behavior, while useful in some situations, was somewhat unexpected. Override the "rootrellink" block in the template to customize where it refers to.
  - The JavaScript search now searches for objects before searching in the full text.
  - TOC tree entries now have CSS classes that make it possible to style them depending on their depth.
  - Highlighted code blocks now have CSS classes that make it possible to style them depending on their language.
  - HTML `<meta>` tags via the docutils `meta` directive are now supported.
  - `SerializingHTMLBuilder` was added as new abstract builder that can be subclassed to serialize build HTML in a specific format. The `PickleHTMLBuilder` is a concrete subclass of it that uses pickle as serialization implementation.
  - `JSONHTMLBuilder` was added as another `SerializingHTMLBuilder` subclass that dumps the generated HTML into JSON files for further processing.

  - The `rellinks` block in the layout template is now called `linktags` to avoid confusion with the relbar links.
  - The HTML builders have two additional attributes now that can be used to disable the anchor-link creation after headlines and definition links.
  - Only generate a module index if there are some modules in the documentation.
- New and changed config values:
  - Added support for internationalization in generated text with the `language` and `locale_dirs` config values. Many thanks to language contributors:
    * Horst Gutmann – German
    * Pavel Kosina – Czech
    * David Larlet – French
    * Michał Kandulski – Polish
    * Yasushi Masuda – Japanese
    * Guillem Borrell – Spanish
    * Luc Saffre and Peter Bertels – Dutch
    * Fred Lin – Traditional Chinese
    * Roger Demetrescu – Brazilian Portuguese
    * Rok Garbas – Slovenian
  - The new config value `highlight_language` set a global default for highlighting. When `'python3'` is selected, console output blocks are recognized like for `'python'`.
  - Exposed Pygments' lexer guessing as a highlight "language" `guess`.
  - The new config value `latex_elements` allows to override all LaTeX snippets that Sphinx puts into the generated .tex file by default.
  - Added `exclude_dirnames` config value that can be used to exclude e.g. CVS directories from source file search.
  - Added `source_encoding` config value to select input encoding.
- Extensions:
  - The new extensions `sphinx.ext.jsmath` and `sphinx.ext.pngmath` provide math support for both HTML and LaTeX builders.
  - The new extension `sphinx.ext.intersphinx` half-automatically creates links to Sphinx documentation of Python objects in other projects.
  - The new extension `sphinx.ext.todo` allows the insertion of "To do" directives whose visibility in the output can be toggled. It also adds a directive to compile a list of all todo items.
  - sphinx.ext.autodoc has a new event `autodoc-process-signature` that allows tuning function signature introspection.
  - sphinx.ext.autodoc has a new event `autodoc-skip-member` that allows tuning which members are included in the generated content.
  - Respect __all__ when autodocumenting module members.
  - The *automodule* directive now supports the `synopsis`, `deprecated` and `platform` options.
- Extension API:

  - `Sphinx.add_node()` now takes optional visitor methods for the HTML, LaTeX and text translators; this prevents having to manually patch the classes.
  - Added `Sphinx.add_javascript()` that adds scripts to load in the default HTML template.
  - Added new events: `source-read`, `env-updated`, `env-purge-doc`, `missing-reference`, `build-finished`.
- Other changes:
  - Added a command-line switch `-Q`: it will suppress warnings.
  - Added a command-line switch `-A`: it can be used to supply additional values into the HTML templates.
  - Added a command-line switch `-C`: if it is given, no configuration file `conf.py` is required.
  - Added a distutils command *build_sphinx*: When Sphinx is installed, you can call `python setup.py build_sphinx` for projects that have Sphinx documentation, which will build the docs and place them in the standard distutils build directory.
  - In quickstart, if the selected root path already contains a Sphinx project, complain and abort.

14.20.2 Bugs fixed

- #51: Escape configuration values placed in HTML templates.
- #44: Fix small problems in HTML help index generation.
- Fix LaTeX output for line blocks in tables.
- #38: Fix "illegal unit" error when using pixel image widths/heights.
- Support table captions in LaTeX output.
- #39: Work around a bug in Jinja that caused "<generator ...>" to be emitted in HTML output.
- Fix a problem with module links not being generated in LaTeX output.
- Fix the handling of images in different directories.
- #29: Support option lists in the text writer. Make sure that dashes introducing long option names are not contracted to en-dashes.
- Support the "scale" option for images in HTML output.
- #25: Properly escape quotes in HTML help attribute values.
- Fix LaTeX build for some description environments with `:noindex:`.
- #24: Don't crash on uncommon casing of role names (like `:Class:`).
- Only output ANSI colors on color terminals.
- Update to newest fncychap.sty, to fix problems with non-ASCII characters at the start of chapter titles.
- Fix a problem with index generation in LaTeX output, caused by hyperref not being included last.
- Don't disregard return annotations for functions without any parameters.
- Don't throw away labels for code blocks.

14.21 Release 0.4.3 (Oct 8, 2008)

- Fix a bug in autodoc with directly given autodoc members.
- Fix a bug in autodoc that would import a module twice, once as "module", once as "module.".
- Fix a bug in the HTML writer that created duplicate `id` attributes for section titles with docutils 0.5.
- Properly call `super()` in overridden blocks in templates.
- Add a fix when using XeTeX.
- Unify handling of LaTeX escaping.
- Rebuild everything when the `extensions` config value changes.
- Don't try to remove a nonexisting static directory.
- Fix an indentation problem in production lists.
- Fix encoding handling for literal include files: `literalinclude` now has an `encoding` option that defaults to UTF-8.
- Fix the handling of non-ASCII characters entered in quickstart.
- Fix a crash with nonexisting image URIs.

14.22 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.

14.23 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.

14.24 Release 0.4 (Jun 23, 2008)

14.24.1 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.

14.24.2 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.

14.25 Release 0.3 (May 6, 2008)

14.25.1 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.

14.25.2 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.

14.26 Release 0.2 (Apr 27, 2008)

14.26.1 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 `block`s were removed; you should use Jinja's standard `{{ super() }}` mechanism instead, as explained in the (newly written) templating docs.

14.26.2 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.

14.26.3 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.

14.27 Release 0.1.61950 (Mar 26, 2008)

- sphinx.quickstart: Fix format string for Makefile.

14.28 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.

14.29 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.

14.30 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.

14.31 Release 0.1.61611 (Mar 21, 2008)

- First public release.

CHAPTER

FIFTEEN

PROJECTS USING SPHINX

This is an (incomplete) alphabetic list of projects that use Sphinx or are experimenting with using it for their documentation. If you like to be included, please mail to the Google group.

I've grouped the list into sections to make it easier to find interesting examples.

15.1 Documentation using the default theme

- APSW: http://apidoc.apsw.googlecode.com/hg/index.html
- ASE: https://wiki.fysik.dtu.dk/ase/
- boostmpi: http://documen.tician.de/boostmpi/
- Calibre: http://calibre.kovidgoyal.net/user_manual/
- CodePy: http://documen.tician.de/codepy/
- Cython: http://docs.cython.org/
- C\C++ Python language binding project: http://language-binding.net/index.html
- Cormoran: http://cormoran.nhopkg.org/docs/
- Director: http://packages.python.org/director/
- Dirigible: http://www.projectdirigible.com/documentation/
- Elemental: http://elemental.googlecode.com/hg/doc/build/html/index.html
- F2py: http://www.f2py.org/html/
- GeoDjango: http://geodjango.org/docs/
- Genomedata: http://noble.gs.washington.edu/proj/genomedata/doc/1.2.2/genomedata.html
- gevent: http://www.gevent.org/
- Google Wave API: http://wave-robot-python-client.googlecode.com/svn/trunk/pydocs/index.html
- GSL Shell: http://www.nongnu.org/gsl-shell/
- Hands-on Python Tutorial: http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/
- Hedge: http://documen.tician.de/hedge/
- Kaa: http://doc.freevo.org/api/kaa/
- Leo: http://webpages.charter.net/edreamleo/front.html

- Lino: http://lino.saffre-rumma.ee/
- MeshPy: http://documen.tician.de/meshpy/
- mpmath: http://mpmath.googlecode.com/svn/trunk/doc/build/index.html
- OpenEXR: http://excamera.com/articles/26/doc/index.html
- OpenGDA: http://www.opengda.org/gdadoc/html/
- openWNS: http://docs.openwns.org/
- Paste: http://pythonpaste.org/script/
- Paver: http://www.blueskyonmars.com/projects/paver/
- Pyccuracy: http://www.pyccuracy.org/
- PyCuda: http://documen.tician.de/pycuda/
- Pyevolve: http://pyevolve.sourceforge.net/
- Pylo: http://documen.tician.de/pylo/
- PyMQI: http://packages.python.org/pymqi/
- PyPubSub: http://pubsub.sourceforge.net/
- pyrticle: http://documen.tician.de/pyrticle/
- Python: http://docs.python.org/
- python-apt: http://apt.alioth.debian.org/python-apt-doc/
- PyUblas: http://documen.tician.de/pyublas/
- Quex: http://quex.sourceforge.net/doc/html/main.html
- Scapy: http://www.secdev.org/projects/scapy/doc/
- Segway: http://noble.gs.washington.edu/proj/segway/doc/1.1.0/segway.html
- SimPy: http://simpy.sourceforge.net/SimPyDocs/index.html
- SymPy: http://docs.sympy.org/
- WTForms: http://wtforms.simplecodes.com/docs/
- z3c: http://docs.carduner.net/z3c-tutorial/

15.2 Documentation using a customized version of the default theme

- Advanced Generic Widgets: http://xoomer.virgilio.it/infinity77/AGW_Docs/index.html
- Bazaar: http://doc.bazaar.canonical.com/en/
- Chaco: http://code.enthought.com/projects/chaco/docs/html/
- Djagios: http://djagios.org/
- GetFEM++: http://home.gna.org/getfem/
- GPAW: https://wiki.fysik.dtu.dk/gpaw/
- Grok: http://grok.zope.org/doc/current/
- IFM: http://fluffybunny.memebot.com/ifm-docs/index.html

- LEPL: http://www.acooke.org/lepl/
- Mayavi: http://code.enthought.com/projects/mayavi/docs/development/html/mayavi
- NOC: http://trac.nocproject.org/trac/wiki/NocGuide
- NumPy: http://docs.scipy.org/doc/numpy/reference/
- Peach^3: http://peach3.nl/doc/latest/userdoc/
- Py on Windows: http://timgolden.me.uk/python-on-windows/
- PyLit: http://pylit.berlios.de/
- Sage: http://sagemath.org/doc/
- SciPy: http://docs.scipy.org/doc/scipy/reference/
- simuPOP: http://simupop.sourceforge.net/manual_release/build/userGuide.html
- Sprox: http://sprox.org/
- TurboGears: http://turbogears.org/2.0/docs/
- Zentyal: http://doc.zentyal.org/
- Zope: http://docs.zope.org/zope2/index.html
- zc.async: http://packages.python.org/zc.async/1.5.0/

15.3 Documentation using the sphinxdoc theme

- Fityk: http://www.unipress.waw.pl/fityk/
- MapServer: http://mapserver.org/
- Matplotlib: http://matplotlib.sourceforge.net/
- Music21: http://mit.edu/music21/doc/html/contents.html
- MyHDL: http://www.myhdl.org/doc/0.6/
- NetworkX: http://networkx.lanl.gov/
- Pweave: http://mpastell.com/pweave/
- Pyre: http://docs.danse.us/pyre/sphinx/
- Pysparse: http://pysparse.sourceforge.net/
- PyTango: http://www.tango-controls.org/static/PyTango/latest/doc/html/index.html
- Reteisi: http://docs.argolinux.org/reteisi/
- Satchmo: http://www.satchmoproject.com/docs/svn/
- Sphinx: http://sphinx.pocoo.org/
- Sqlkit: http://sqlkit.argolinux.org/
- Tau: http://www.tango-controls.org/static/tau/latest/doc/html/index.html
- Total Open Station: http://tops.berlios.de/
- WebFaction: http://docs.webfaction.com/

15.4 Documentation using another builtin theme

- C/C++ Development with Eclipse: http://book.dehlia.in/c-cpp-eclipse/ (agogo)
- Distribute: http://packages.python.org/distribute/ (nature)
- Jinja: http://jinja.pocoo.org/2/documentation/ (scrolls)
- jsFiddle: http://doc.jsfiddle.net/ (nature)
- pip: http://pip.openplans.org/ (nature)
- Programmieren mit PyGTK und Glade (German): http://www.florian-diesch.de/doc/python-und-glade/online/ (agogo)
- Spring Python: http://springpython.webfactional.com/current/sphinx/index.html (nature)
- sqlparse: http://python-sqlparse.googlecode.com/svn/docs/api/index.html (agogo)
- Sylli: http://sylli.sourceforge.net/ (nature)
- libLAS: http://liblas.org/ (nature)

15.5 Documentation using a custom theme/integrated in a site

- Blender: http://www.blender.org/documentation/250PythonDoc/
- Blinker: http://discorporate.us/projects/Blinker/docs/
- Classy: classy: http://classy.pocoo.org/
- Django: http://docs.djangoproject.com/
- e-cidadania: http://e-cidadania.readthedocs.org/en/latest/
- Flask: http://flask.pocoo.org/docs/
- Flask-OpenID: http://packages.python.org/Flask-OpenID/
- Gameduino: http://excamera.com/sphinx/gameduino/
- GeoServer: http://docs.geoserver.org/
- Glashammer: http://glashammer.org/
- MirrorBrain: http://mirrorbrain.org/docs/
- nose: http://somethingaboutorange.com/mrl/projects/nose/
- ObjectListView: http://objectlistview.sourceforge.net/python
- Open ERP: http://doc.openerp.com/
- OpenLayers: http://docs.openlayers.org/
- PyEphem: http://rhodesmill.org/pyephem/
- German Plone 4.0 user manual: http://www.hasecke.com/plone-benutzerhandbuch/4.0/
- Pylons: http://pylonshq.com/docs/en/0.9.7/
- PyMOTW: http://www.doughellmann.com/PyMOTW/
- pypol: http://pypol.altervista.org/ (celery)
- qooxdoo: http://manual.qooxdoo.org/current

- Roundup: http://www.roundup-tracker.org/
- Selenium: http://seleniumhq.org/docs/
- Self: http://selflanguage.org/
- SQLAlchemy: http://www.sqlalchemy.org/docs/
- tinyTiM: http://tinytim.sourceforge.net/docs/2.0/
- tipfy: http://www.tipfy.org/docs/
- Werkzeug: http://werkzeug.pocoo.org/documentation/dev/
- WFront: http://discorporate.us/projects/WFront/

15.6 Homepages and other non-documentation sites

- Applied Mathematics at the Stellenbosch University: http://dip.sun.ac.za/
- A personal page: http://www.dehlia.in/
- Benoit Boissinot: http://perso.ens-lyon.fr/benoit.boissinot/
- lunarsite: http://lunaryorn.de/
- Red Hot Chili Python: http://redhotchilipython.com/
- The Wine Cellar Book: http://www.thewinecellarbook.com/doc/en/
- VOR: http://www.vor-cycling.be/

15.7 Books produced using Sphinx

- "The `repoze.bfg` Web Application Framework": http://www.amazon.com/repoze-bfg-Web-Application-Framework-Version/dp/0615345379
- A Theoretical Physics Reference book: http://theoretical-physics.net/
- "Simple and Steady Way of Learning for Software Engineering" (in Japanese): http://www.amazon.co.jp/dp/477414259X/
- "Expert Python Programming" (Japanese translation): http://www.amazon.co.jp/dp/4048686291/
- "Pomodoro Technique Illustrated" (Japanese translation): http://www.amazon.co.jp/dp/4048689525/

PYTHON MODULE INDEX

a

b

c

d

e