Sphinx Documentation

Release 0.5.2

Georg Brandl

May 03, 2009

Contents

CHAPTER 1

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.

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

1.1 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 Launchpad.
- 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.

1.2 Prerequisites

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

1.3 Setting up the documentation sources

The root directory of a documentation collection is called the *source directory*. Normally, this directory also contains the Sphinx configuration file `conf.py`, but that file can also live in another directory, the

configuration directory. New in version 0.3: Support for a different configuration directory. Sphinx comes with a script called **sphinx-quickstart** that sets up a source directory and creates a default `conf.py` from a few questions it asks you. Just run

```
$ sphinx-quickstart
```

and answer the questions.

1.4 Running a build

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

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

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

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

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

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

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

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

-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 value.)

-A *name=value* Make the *name* assigned to *value* in the HTML templates.

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

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

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

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

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

CHAPTER 2

Sphinx concepts

2.1 Document names

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

2.2 The TOC tree

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

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

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

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

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

This accomplishes two things:

[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}`.

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

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

```
.. toctree::

   intro
   All about strings <strings>
   datatypes
```

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

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

```
.. toctree::
   :glob:

   intro*
   recipe/*
   *
```

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

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

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

2.3 Special names

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

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

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

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

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

[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.

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

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

- every name beginning with _

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

CHAPTER 3

reStructuredText Primer

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

See Also:

The authoritative reStructuredText User Documentation.

3.1 Paragraphs

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

3.2 Inline markup

The standard reST inline markup is quite simple: use

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

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

Be aware of some restrictions of this markup:

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

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

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

3.3 Lists and Quotes

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

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

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

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

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

```
* this is
* a list

  * with a nested list
  * and some subitems

* and here the parent list continues
```

Definition lists are created as follows:

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

   and can even consist of multiple paragraphs

next term
   Description.
```

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

3.4 Source Code

Literal code blocks are introduced by ending a paragraph with the special marker `::`. The literal block must be indented (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:".

3.5 Hyperlinks

3.5.1 External links

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

3.5.2 Internal links

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

3.6 Sections

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

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

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

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

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

3.7 Explicit Markup

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

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

3.8 Directives

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

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

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

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

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

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

3.9 Images

reST supports an image directive, used like so:

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

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

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.

3.10 Footnotes

For footnotes, 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 (`[#]_`).

3.11 Citations

Standard reST citations 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 `#`.

3.12 Substitutions

reST supports "substitutions", 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*
```

See the reST reference for substitutions for details.

If you want to use some substitutions for all documents, 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*.

3.13 Comments

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

3.14 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.

3.15 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.
- **No nested inline markup:** Something like `*see :func:`foo`*` is not possible.

CHAPTER 4

Sphinx Markup Constructs

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

4.1 Module-specific markup

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

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

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

The directives you can use for module declarations are:

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

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

The `platform` option, if present, is a comma-separated list of the platforms on which the module is available (if it is available on all platforms, the option should be omitted). The keys are short identifiers; examples that are in use include "IRIX", "Mac", "Windows", and "Unix". It is important to use a key which has already been used when applicable.

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

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

.. **currentmodule:: name**
This directive tells Sphinx that the classes, functions etc. documented from here are in the given mod-

ule (like `module`), but it will not create index entries, an entry in the Global Module Index, or a link target for `mod`. This is helpful in situations where documentation for things in a module is spread over multiple files or sections – one location has the `module` directive, the others only `currentmodule`.

`..` **`moduleauthor:: name <email>`**
The `moduleauthor` directive, which can appear multiple times, names the authors of the module code, just like `sectionauthor` names the author(s) of a piece of documentation. It too only produces output if the `show_authors` configuration value is True.

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

4.2 Object description units

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

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

   Spam or ham the foo.
```

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

The directives are:

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

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

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

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

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

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

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

`..` **`cmacro:: name`**
Describes a "simple" C macro. Simple macros are macros which are used for code expansion, but which do not take arguments so cannot be described as functions. This is not to be used for simple

constant definitions. Examples of its use in the Python documentation include `PyObject_HEAD` and `Py_BEGIN_ALLOW_THREADS`.

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

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

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

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

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

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

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

Object methods are not documented using this directive. Bound object methods placed in the module namespace as part of the public interface of the module are documented using this, as they are equivalent to normal functions for most purposes.

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

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

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

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

-- or --

.. class:: Bar

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

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

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

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

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

4.2.1 Signatures

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

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

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

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

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

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

4.2.2 Info field lists

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

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

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

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

   Format the exception with a traceback.

   :param 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

4.2.3 Command-line program markup

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

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

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

   Run a module as a script.
```

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

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

.. **program:: name**
Like `currentmodule`, this directive produces no output. Instead, it serves to notify Sphinx that all following `cmdoption` 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

.. cmdoption:: -r

   Work recursively.

.. program:: svn

.. cmdoption:: -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.

4.2.4 Custom description units

There is also a generic version of these directives:

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

```
.. describe:: opcode

   Describes a Python bytecode instruction.
```

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

4.3 Paragraph-level markup

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

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

Example:

```
.. note::

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

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

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

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

Example:

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

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

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

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

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

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

```
.. seealso::

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

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

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

```
.. seealso:: modules :mod:`zipfile`, :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", 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
```

4.4 Table-of-contents markup

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

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

4.5 Index-generating markup

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

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

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

For example:

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

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

...
```

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

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

The possible entry types are:

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

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

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

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

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

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

This creates four index entries.

4.6 Glossary

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

```
.. glossary::

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

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

4.7 Grammar production displays

Special markup is available for displaying the productions of a formal grammar. The markup is simple and does not attempt to model all aspects of BNF (or any derived forms), but provides enough to allow

context-free grammars to be displayed in a way that causes uses of a symbol to be rendered as hyperlinks to the definition of the symbol. There is this directive:

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

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

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

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

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

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

4.8 Showing code examples

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

Representing an interactive session requires including the prompts and output along with the Python code. No special markup is required for interactive sessions. After the last line of input or output presented, there should not be an "unused" primary prompt; this is an example of what *not* to do:

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

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

- There is a "highlighting language" for each source file. Per default, this is `'python'` as the majority of files will have to highlight Python snippets, 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.

4.8.1 Line numbers

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

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

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

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

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

   Some more Ruby code.
```

4.8.2 Includes

`..` **`literalinclude::`** **`filename`**

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

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

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

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

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

```
.. 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
```

New in version 0.4.3: The `encoding` option.

4.9 Inline markup

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

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

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

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

4.9.1 Cross-referencing syntax

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

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

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

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

4.9.2 Cross-referencing Python objects

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

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

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

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

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

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

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

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

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

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

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

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

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

4.9.3 Cross-referencing C constructs

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

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

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

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

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

4.9.4 Cross-referencing other items of interest

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

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

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

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

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

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

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

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

4.9.5 Cross-referencing arbitrary locations

To support cross-referencing to arbitrary locations in 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".

- 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.

4.9.6 Other semantic markup

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

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

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

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

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

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

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

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

:mailheader:
The name of an RFC 822-style mail header. This markup does not imply that the header is being used in an email message, but can be used to refer to any header of the same "style." This is also used for headers defined by the various MIME specifications. The header name should be entered in the same way it would normally be found in practice, with the camel-casing conventions being preferred where there is more than one common usage. For example: `:mailheader:`Content-Type``.

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

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

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

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

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

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

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

:newsgroup:
The name of a Usenet newsgroup.

:program:
The name of an executable program. This may differ from the file name for the executable for some platforms. In particular, the `.exe` (or other) extension should be omitted for Windows programs.

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

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

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

The following roles generate external links:

:pep:
A reference to a Python Enhancement Proposal. This generates appropriate index entries. The text "PEP *number*" is generated; in the HTML output, this text is a hyperlink to an online copy of the specified PEP.

:rfc:
A reference to an Internet Request for Comments. This generates appropriate index entries. The text "RFC *number*" is generated; in the HTML output, this text is a hyperlink to an online copy of the specified RFC.

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

4.9.7 Substitutions

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

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

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

|today|
Replaced by either today's date (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`.

4.10 Miscellaneous markup

4.10.1 File-wide metadata

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

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

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

At the moment, these metadata fields are recognized:

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

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

4.10.2 Meta-information markup

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

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

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

4.10.3 Tables

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

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

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

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

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

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

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

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

CHAPTER 5

Available builders

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

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

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

Its name is `html`.

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

Its name is `htmlhelp`.

class **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`.

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

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

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

Its name is `changes`.

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

Its name is `linkcheck`.

Built-in Sphinx extensions that offer more builders are:

- doctest
- coverage

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

last_updated Date of last build.

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

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

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. (XXX: document common members)

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

CHAPTER 6

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

6.1 General configuration

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

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

exclude_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.

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.

exclude_dirs
A list of directory names, relative to the source directory, that are to be excluded from the search for source files. Deprecated since version 0.5: This does not take subdirs of the excluded directories into account. Use `exclude_trees` or `exclude_dirnames`, which match the expectations.

locale_dirs
New in version 0.5. Directories in which to search for additional Sphinx message catalogs (see `language`), relative to the source directory. The directories on this path are searched by the standard `gettext` module for a domain of `sphinx`; so if you add the directory `./locale` to this settting, the message catalogs 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 templates). Relative paths are taken as relative to the configuration directory.

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

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

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

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

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.

6.2 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:

- `cs` – Czech
- `de` – German
- `en` – English
- `es` – Spanish
- `fr` – French
- `nl` – Dutch
- `pl` – Polish
- `pt_BR` – Brazilian Portuguese
- `sl` – Slovenian
- `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. Default is `'sphinx'`, which is a builtin style designed to match Sphinx' default style. Changed in version 0.3: If the value is a fully-qualified name of a custom Pygments style class, this is then used as custom style.

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

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

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

6.3 Options for HTML output

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

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

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

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

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

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

html_static_path
A list of paths that contain custom static files (such as style sheets or script files). Relative paths are taken as relative to the configuration directory. They are copied to the output directory after the

builtin static files, so a file named `default.css` will overwrite the builtin `default.css`. Changed in version 0.4: The paths in `html_static_path` can now contain subdirectories.

html_last_updated_fmt
If this is not the empty string, a 'Last updated on:' timestamp is inserted at every page bottom, using the given `strftime()` format. Default is `'%b %d, %Y'` (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_sidebars
Custom sidebar templates, must be a dictionary that maps document names to template names. Example:

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

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

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

Example:

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

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

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

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

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

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

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

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

> **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_use_opensearch
If nonempty, an *OpenSearch <http://opensearch.org>* description file will be output, and all pages will contain a `<link>` tag referring to it. Since OpenSearch doesn't support relative URLs for its search

page location, the value of this option must be the base URL from which these documents are served (without trailing slash), e.g. `"http://docs.python.org"`. The default is `''`.

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

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

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

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

6.4 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*: Must be one of `'manual'` or `'howto'`. Only "manual" documents will get appendices. Also, howtos will have a simpler title page.
- *toctree_only*: Must be `True` or `False`. If `True`, the *startdoc* document itself is not included in the output, only the documents referenced by it via TOC trees. With this option, you can put extra stuff in the master document that shows up in the HTML, but not the LaTeX output.

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

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

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

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

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

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'` `'makemodindex'` `'shorthandoff'` `'printmodindex'`

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.

CHAPTER 7

Templating

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

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

No. You have several other options:

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

7.2 Jinja/Sphinx Templating Primer

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

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

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

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

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

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

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

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

7.2.1 Blocks

The following blocks exist in the `layout` template:

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

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 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.

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

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

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

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

sidebartoc The table of contents within the sidebar.

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

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

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

7.2.2 Configuration Variables

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

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

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

Overriding works like this:

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

7.2.3 Helper Functions

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

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

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

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

sidebar()
Return the rendered sidebar.

relbar()
Return the rendered relbar.

7.2.4 Global Variables

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

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

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

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

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

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

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

CHAPTER 8

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.

8.1 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.)

8.1.1 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.

8.1.2 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).

8.1.3 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', todo_directive, 1, (0, 0, 1))
    app.add_directive('todolist', todolist_directive, 0, (0, 0, 0))
    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, handler function and two arguments that specify if the directive has content and how many arguments it accepts.

  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.

8.1.4 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.

8.1.5 The Directive Handlers

A directive handler is a function with a host of arguments, covered in detail in the docutils documentation. It must return a list of nodes.

The `todolist` directive is quite simple:

```
def todolist_directive(name, arguments, options, content, lineno,
                       content_offset, block_text, state, state_machine):
    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

def todo_directive(name, arguments, options, content, lineno,
                   content_offset, block_text, state, state_machine):
    env = state.document.settings.env

    targetid = "todo-%s" % env.index_num
    env.index_num += 1
    targetnode = nodes.target('', '', ids=[targetid])

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

    if not hasattr(env, 'todo_all_todos'):
        env.todo_all_todos = []
    env.todo_all_todos.append({
        'docname': env.docname,
        'lineno': 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 `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.index_num` which is persistent between directive calls 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.

8.1.6 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):
            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.

8.2 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:

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

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

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

add_node(*node, **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 and text writers can be given as keyword arguments: the keyword must be one or more of `'html'`, `'latex'`, `'text'`, 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)

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.

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

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)
```

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

add_description_unit(*directivename, rolename, indextemplate='', parse_node=None, ref_nodeclass=None*)
This method is a very convenient way to add a new type of information that can be cross-referenced. It will do this:

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

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

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

you can use this markup in your documents:

```
.. directive:: function

   Document a function.

<...>

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

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

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

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

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

add_crossref_type(*directivename*, *rolename*, *indextemplate=''*, *ref_nodeclass=None*)
This method is very similar to `add_description_unit()` except that the directive it generates must be empty, and will produce no output.

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

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

Example usage:

```
.. topic:: application API

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

<...>

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

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

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

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`. New in version 0.5.

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

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

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

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

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` (and don't call the rest of the callbacks). New in version 0.5.

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

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

8.2.1 Sphinx core events

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

builder-initedapp
Emitted when the builder object has been created. It is available as `app.builder`.

env-purge-docapp, 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-readapp, 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-readapp, 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-referenceapp, 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-resolvedapp, 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-updatedapp, 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.

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

The *pagename* argument is the canonical name of the page being rendered, that is, without `.html` suffix and using slashes as path separators. The *templatename* is the name of the template to render,

this will be `'page.html'` for all pages from reST documents.

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

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

build-finishedapp, 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.

8.2.2 The template bridge

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

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

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

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

8.3 Writing new builders

Todo

Expand this.

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

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

load_env()
Set up the build environment.

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

build_all()
Build all source files.

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

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

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

These methods must be overridden in concrete builder classes:

init()
Load necessary templates and perform initialization.

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

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

prepare_writing(*docnames*)

write_doc(*docname, doctree*)

finish()

8.4 Builtin Sphinx extensions

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

8.4.1 `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 `module`, `class` and so forth. On parsing time, they import the corresponding module and extract the docstring of the given objects, inserting them into the page source under a suitable `module`, `class` etc. directive.

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

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

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

will produce source like this:

```
.. class:: Noodle

   Noodle's docstring.
```

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

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

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

   .. method:: boil(time=10)

      Boil the noodle *time* minutes.
```

Options and advanced usage

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

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

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

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

  will document exactly the specified members.
- Members without docstrings will be left out, unless you give the `undoc-members` flag option:

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

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

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

  This can be combined with `undoc-members` to document *all* available members of the class or module. New in version 0.3.
- It's possible to override the signature for callable members (functions, methods, classes) with the regular syntax that will override the signature gained from instropection:

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

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

  This is useful if the signature from the method is hidden by a decorator. New in version 0.4.
- The `autoclass` and `autoexception` directives also support a flag option called `show-inheritance`. When given, a list of base classes will be inserted just below the class signature. New in version 0.4.
- All autodoc directives support the `noindex` flag option that has the same effect as for standard `function` etc. directives: no index entries are generated for the documented object (and all autodocumented members). New in version 0.4.
- `automodule` also recognizes the `synopsis`, `platform` and `deprecated` options that the standard `module` directive supports. New in version 0.5.

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

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

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

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

There are also new config values that you can set:

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

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

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

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

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

New in version 0.3.

Docstring preprocessing

autodoc provides the following additional events:

autodoc-process-docstringapp, 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-signatureapp, 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`:

cut_lines(*pre, post=0, what=None*)

Return a listener that removes the first *pre* and last *post* lines of every docstring. If *what* is a sequence of strings, only docstrings of a type in *what* will be processed.

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

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

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

between(*marker, what=None, keepempty=False*)

Return a listener that only keeps lines between lines that match the *marker* regular expression. If no line matches, the resulting docstring would be empty, so no change will be made unless *keepempty* is true.

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

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

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

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

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

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

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

There are two kinds of test blocks:

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

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

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

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

This directive supports two options:

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

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

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

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

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

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

This directive supports one option:

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

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

This directive supports two options:

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

Example:

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

   Output text.
```

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

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

.. testsetup:: *

   import parrot

The parrot module is a module about parrots.

Doctest example:

.. doctest::

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

Test-Output example:

.. testcode::

   parrot.voom(3000)

This would output:

.. testoutput::

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

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

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

doctest_test_doctest_blocks

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

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

```
Some documentation text.

>>> print 1
1

Some more documentation text.
```

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

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

```
Some documentation text.

.. doctest::

   >>> print 1
   1

Some more documentation text.
```

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

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

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

New in version 0.5. This extension can generate automatic links to the documentation of Python objects in other projects. This works as follows:

- Each Sphinx HTML build creates a file named `objects.inv` that contains a mapping from Python identifiers 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 Python 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

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.

Relative local paths in the keys are taken as relative to the base of the built documentation, while relative local paths in the values are taken as relative to the source directory.

An example, to add links to modules and objects in the Python standard library documentation:

```
intersphinx_mapping = {'http://docs.python.org/dev': 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:

```
intersphinx_mapping = {'http://docs.python.org/dev': 'python-inv.txt'}
```

This will read the inventory from `python.inv` in the source directory, but still generate links to the pages under `http://docs.python.org/dev`. 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.

8.4.4 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: `sphinx.ext.mathbase` does not need to be added to the `extensions` config value.

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` 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 root of the built docs.

For example, if you put JSMath into the static path of the Sphinx docs, this value would be `_static/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.

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

Todo

Write this section.

8.4.6 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).

8.4.7 sphinx.ext.coverage – Collect doc coverage stats

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

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

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

8.4.8 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`.

8.5 Third-party extensions

There are several extensions that are not (yet) maintained in the Sphinx distribution. The Wiki at BitBucket maintains a list of those.

If you write an extension that you think others will find useful, please write to the project mailing list (sphinx-dev@googlegroups.com) and we'll find the proper way of including or hosting it for the public.

8.5.1 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 9

Glossary

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

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

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

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

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

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

CHAPTER 10

Changes in Sphinx

10.1 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().

10.2 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.

10.3 Release 0.5 (Nov 23, 2008) – Birthday release!

10.3.1 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.

10.3.2 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.

10.4 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.

10.5 Release 0.4.2 (Jul 29, 2008)

- Fix rendering of the `samp` role in HTML.
- Fix a bug with LaTeX links to headings leading to a wrong page.
- Reread documents with globbed toctrees when source files are added or removed.
- Add a missing parameter to PickleHTMLBuilder.handle_page().
- Put inheritance info always on its own line.
- Don't automatically enclose code with whitespace in it in quotes; only do this for the `samp` role.

- autodoc now emits a more precise error message when a module can't be imported or an attribute can't be found.
- The JavaScript search now uses the correct file name suffix when referring to found items.
- The automodule directive now accepts the `inherited-members` and `show-inheritance` options again.
- You can now rebuild the docs normally after relocating the source and/or doctree directory.

10.6 Release 0.4.1 (Jul 5, 2008)

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

10.7 Release 0.4 (Jun 23, 2008)

10.7.1 New features added

- `tocdepth` can be given as a file-wide metadata entry, and specifies the maximum depth of a TOC of this file.
- The new config value *default_role* can be used to select the default role for all documents.
- Sphinx now interprets field lists with fields like `:param foo:` in description units.
- The new *staticmethod* directive can be used to mark methods as static methods.
- HTML output:
  - The "previous" and "next" links have a more logical structure, so that by following "next" links you can traverse the entire TOC tree.
  - The new event *html-page-context* can be used to include custom values into the context used when rendering an HTML template.
  - Document metadata is now in the default template context, under the name *metadata*.

  - The new config value *html_favicon* can be used to set a favicon for the HTML output. Thanks to Sebastian Wiesner.
  - The new config value *html_use_index* can be used to switch index generation in HTML documents off.
  - The new config value *html_split_index* can be used to create separate index pages for each letter, to be used when the complete index is too large for one page.
  - The new config value *html_short_title* can be used to set a shorter title for the documentation which is then used in the navigation bar.
  - The new config value *html_show_sphinx* can be used to control whether a link to Sphinx is added to the HTML footer.
  - The new config value *html_file_suffix* can be used to set the HTML file suffix to e.g. `.xhtml`.
  - The directories in the *html_static_path* can now contain subdirectories.
  - The module index now isn't collapsed if the number of submodules is larger than the number of toplevel modules.

- The image directive now supports specifying the extension as `.*`, which makes the builder select the one that matches best. Thanks to Sebastian Wiesner.
- The new config value *exclude_trees* can be used to exclude whole subtrees from the search for source files.
- Defaults for configuration values can now be callables, which allows dynamic defaults.
- The new TextBuilder creates plain-text output.
- Python 3-style signatures, giving a return annotation via `->`, are now supported.
- Extensions:
  - The autodoc extension now offers a much more flexible way to manipulate docstrings before including them into the output, via the new *autodoc-process-docstring* event.
  - The *autodoc* extension accepts signatures for functions, methods and classes now that override the signature got via introspection from Python code.
  - The *autodoc* extension now offers a `show-inheritance` option for autoclass that inserts a list of bases after the signature.
  - The autodoc directives now support the `noindex` flag option.

10.7.2 Bugs fixed

- Correctly report the source location for docstrings included with autodoc.
- Fix the LaTeX output of description units with multiple signatures.
- Handle the figure directive in LaTeX output.
- Handle raw admonitions in LaTeX output.
- Fix determination of the title in HTML help output.
- Handle project names containing spaces.
- Don't write SSI-like comments in HTML output.
- Rename the "sidebar" class to "sphinxsidebar" in order to stay different from reST sidebars.

- Use a binary TOC in HTML help generation to fix issues links without explicit anchors.
- Fix behavior of references to functions/methods with an explicit title.
- Support citation, subscript and superscript nodes in LaTeX writer.
- Provide the standard "class" directive as "cssclass"; else it is shadowed by the Sphinx-defined directive.
- Fix the handling of explicit module names given to autoclass directives. They now show up with the correct module name in the generated docs.
- Enable autodoc to process Unicode docstrings.
- The LaTeX writer now translates line blocks with `\raggedright`, which plays nicer with tables.
- Fix bug with directories in the HTML builder static path.

10.8 Release 0.3 (May 6, 2008)

10.8.1 New features added

- The `toctree` directive now supports a `glob` option that allows glob-style entries in the content.
- If the *pygments_style* config value contains a dot it's treated as the import path of a custom Pygments style class.
- A new config value, *exclude_dirs*, can be used to exclude whole directories from the search for source files.
- The configuration directory (containing `conf.py`) can now be set independently from the source directory. For that, a new command-line option `-c` has been added.
- A new directive `tabularcolumns` can be used to give a tabular column specification for LaTeX output. Tables now use the `tabulary` package. Literal blocks can now be placed in tables, with several caveats.
- A new config value, *latex_use_parts*, can be used to enable parts in LaTeX documents.
- Autodoc now skips inherited members for classes, unless you give the new `inherited-members` option.
- A new config value, *autoclass_content*, selects if the docstring of the class' `__init__` method is added to the directive's body.
- Support for C++ class names (in the style `Class::Function`) in C function descriptions.
- Support for a `toctree_only` item in items for the `latex_documents` config value. This only includes the documents referenced by TOC trees in the output, not the rest of the file containing the directive.

10.8.2 Bugs fixed

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

10.9 Release 0.2 (Apr 27, 2008)

10.9.1 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.

10.9.2 New features added

- Extension API (Application object):
  - Support a new method, `add_crossref_type`. It works like `add_description_unit` but the directive will only create a target and no output.
  - Support a new method, `add_transform`. It takes a standard docutils `Transform` subclass which is then applied by Sphinx' reader on parsing reST document trees.
  - Add support for other template engines than Jinja, by adding an abstraction called a "template bridge". This class handles rendering of templates and can be changed using the new configuration value "template_bridge".
  - The config file itself can be an extension (if it provides a `setup()` function).

- Markup:
  - New directive, `currentmodule`. It can be used to indicate the module name of the following documented things without creating index entries.
  - Allow giving a different title to documents in the toctree.
  - Allow giving multiple options in a `cmdoption` directive.
  - Fix display of class members without explicit class name given.
- Templates (HTML output):
  - `index.html` renamed to `defindex.html`, see above.
  - There's a new config value, `html_title`, that controls the overall "title" of the set of Sphinx docs. It is used instead everywhere instead of "Projectname vX.Y documentation" now.
  - All references to "documentation" in the templates have been removed, so that it is now easier to use Sphinx for non-documentation documents with the default templates.
  - Templates now have an XHTML doctype, to be consistent with docutils' HTML output.
  - You can now create an OpenSearch description file with the `html_use_opensearch` config value.
  - You can now quickly include a logo in the sidebar, using the `html_logo` config value.
  - There are new blocks in the sidebar, so that you can easily insert content into the sidebar.
- LaTeX output:
  - The `sphinx.sty` package was cleaned of unused stuff.
  - You can include a logo in the title page with the `latex_logo` config value.
  - You can define the link colors and a border and background color for verbatim environments.

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

10.9.3 Bugs fixed

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

10.10 Release 0.1.61950 (Mar 26, 2008)

- sphinx.quickstart: Fix format string for Makefile.

10.11 Release 0.1.61945 (Mar 26, 2008)

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

10.12 Release 0.1.61843 (Mar 24, 2008)

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

10.13 Release 0.1.61798 (Mar 23, 2008)

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

- sphinx.ext.autodoc: Render `.. automodule::` docstrings in a section node, so that module docstrings can contain proper sectioning.
- sphinx.ext.autodoc: Use the module's encoding for decoding docstrings, rather than requiring ASCII.

10.14 Release 0.1.61611 (Mar 21, 2008)

- First public release.

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.

- APSW: http://apsw.googlecode.com/svn/publish/index.html
- Calibre: http://calibre.kovidgoyal.net/user_manual/
- Chaco: http://code.enthought.com/projects/chaco/docs/html/
- Cython: http://docs.cython.org/
- Director: http://packages.python.org/director/
- Django: http://docs.djangoproject.com/
- F2py: http://www.f2py.org/html/
- GeoDjango: http://geodjango.org/docs/
- Glashammer: http://glashammer.org/
- Grok: http://grok.zope.org/doc/current/
- IFM: http://fluffybunny.memebot.com/ifm-docs/index.html
- Jinja: http://jinja.pocoo.org/2/documentation/
- Matplotlib: http://matplotlib.sourceforge.net/
- Mayavi: http://code.enthought.com/projects/mayavi/docs/development/html/mayavi
- Mixin.com: http://dev.mixin.com/
- NetworkX: http://networkx.lanl.gov/
- NumPy: http://docs.scipy.org/doc/numpy/reference/
- ObjectListView: http://objectlistview.sourceforge.net/python
- Paste: http://pythonpaste.org/script/
- Paver: http://www.blueskyonmars.com/projects/paver/
- Py on Windows: http://timgolden.me.uk/python-on-windows/

- PyEphem: http://rhodesmill.org/pyephem/
- PyPubSub: http://pubsub.sourceforge.net/
- PyUblas: http://tiker.net/doc/pyublas/
- Pylons: http://docs.pylonshq.com/
- Pysparse: http://pysparse.sourceforge.net/
- Python: http://docs.python.org/dev/
- Satchmo: http://www.satchmoproject.com/docs/svn/
- Sphinx: http://sphinx.pocoo.org/
- SQLAlchemy: http://www.sqlalchemy.org/docs/
- SymPy: http://docs.sympy.org/
- tinyTiM: http://tinytim.sourceforge.net/docs/2.0/
- TurboGears: http://turbogears.org/2.0/docs/
- Zope 3: e.g. http://docs.carduner.net/z3c-tutorial/
- mpmath: http://mpmath.googlecode.com/svn/trunk/doc/build/index.html
- zc.async: http://packages.python.org/zc.async/1.5.0/

Module Index

C

S

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W