
Documenting Python

Release 2.7

Georg Brandl

August 26, 2010

Python Software Foundation
Email: docs@python.org

CONTENTS

1	Introduction	3
2	Style Guide	5
3	reStructuredText Primer	7
3.1	Paragraphs	7
3.2	Inline markup	7
3.3	Lists and Quotes	8
3.4	Source Code	8
3.5	Hyperlinks	9
3.6	Sections	9
3.7	Explicit Markup	9
3.8	Directives	10
3.9	Footnotes	10
3.10	Comments	10
3.11	Source encoding	10
3.12	Gotchas	10
4	Additional Markup Constructs	13
4.1	Meta-information markup	13
4.2	Module-specific markup	13
4.3	Information units	14
4.4	Showing code examples	16
4.5	Inline markup	17
4.6	Cross-linking markup	20
4.7	Paragraph-level markup	20
4.8	Table-of-contents markup	22
4.9	Index-generating markup	22
4.10	Grammar production displays	23
4.11	Substitutions	23
5	Differences to the LaTeX markup	25
5.1	Inline markup	25
5.2	Information units	26
5.3	Structure	27
6	Building the documentation	29
6.1	Using make	29
6.2	Without make	30

A	Glossary	31
B	About these documents	39
B.1	Contributors to the Python Documentation	39
C	History and License	41
C.1	History of the software	41
C.2	Terms and conditions for accessing or otherwise using Python	42
C.3	Licenses and Acknowledgements for Incorporated Software	44
D	Copyright	53
	Index	55

The Python language has a substantial body of documentation, much of it contributed by various authors. The markup used for the Python documentation is `reStructuredText`, developed by the `docutils` project, amended by custom directives and using a toolset named `Sphinx` to postprocess the HTML output.

This document describes the style guide for our documentation as well as the custom `reStructuredText` markup introduced by `Sphinx` to support Python documentation and how it should be used.

Note: If you're interested in contributing to Python's documentation, there's no need to write `reStructuredText` if you're not so inclined; plain text contributions are more than welcome as well. Send an e-mail to docs@python.org or open an issue on the *tracker*.

INTRODUCTION

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

The involvement of the community takes many forms, from authoring to bug reports to just plain complaining when the documentation could be more complete or easier to use.

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

If your interest is in contributing to the Python documentation, but you don't have the time or inclination to learn reStructuredText and the markup structures documented here, there's a welcoming place for you among the Python contributors as well. Any time you feel that you can clarify existing documentation or provide documentation that's missing, the existing documentation team will gladly work with you to integrate your text, dealing with the markup for you. Please don't let the material in this document stand between the documentation and your desire to help out!

STYLE GUIDE

The Python documentation should follow the [Apple Publications Style Guide](#) wherever possible. This particular style guide was selected mostly because it seems reasonable and is easy to get online.

Topics which are not covered in Apple’s style guide will be discussed in this document.

All reST files use an indentation of 3 spaces. The maximum line length is 80 characters for normal text, but tables, deeply indented code samples and long links may extend beyond that.

Make generous use of blank lines where applicable; they help grouping things together.

A sentence-ending period may be followed by one or two spaces; while reST ignores the second space, it is customarily put in by some users, for example to aid Emacs’ auto-fill mode.

Footnotes are generally discouraged, though they may be used when they are the best way to present specific information. When a footnote reference is added at the end of the sentence, it should follow the sentence-ending punctuation. The reST markup should appear something like this:

```
This sentence has a footnote reference. [#]_ This is the next sentence.
```

Footnotes should be gathered at the end of a file, or if the file is very long, at the end of a section. The docutils will automatically create backlinks to the footnote reference.

Footnotes may appear in the middle of sentences where appropriate.

Many special names are used in the Python documentation, including the names of operating systems, programming languages, standards bodies, and the like. Most of these entities are not assigned any special markup, but the preferred spellings are given here to aid authors in maintaining the consistency of presentation in the Python documentation.

Other terms and words deserve special mention as well; these conventions should be used to ensure consistency throughout the documentation:

CPU For “central processing unit.” Many style guides say this should be spelled out on the first use (and if you must use it, do so!). For the Python documentation, this abbreviation should be avoided since there’s no reasonable way to predict which occurrence will be the first seen by the reader. It is better to use the word “processor” instead.

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

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

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

Unix The name of the operating system developed at AT&T Bell Labs in the early 1970s.

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: `this is\ *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:

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-linking markup*.

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, we use this convention:

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

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 Footnotes

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

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

```
.. rubric:: Footnotes

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

You can also explicitly number the footnotes for better context.

3.10 Comments

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

3.11 Source encoding

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

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

3.12 Gotchas

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

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

ADDITIONAL 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. Documentation for “standard” reST constructs is not included here, though they are used in the Python documentation.

Note: This is just an overview of Sphinx’ extended markup capabilities; full coverage can be found in [its own documentation](#).

4.1 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 (though it isn’t) and email address. The domain name portion of the address should be lower case. Example:

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

Currently, this markup isn’t reflected in the output in any way, but it helps keep track of contributions.

4.2 Module-specific markup

The markup described in this section is used to provide information about a module being documented. Each module should be documented in its own file. Normally this markup appears after the title heading of that file; a typical file 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>
```

As you can see, the module-specific markup consists of two directives, the `module` directive and the `moduleauthor` directive.

module

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).

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.

moduleauthor

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 does not result in any output currently.

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.

4.3 Information 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

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

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

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

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

cvar

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

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

data

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

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

function

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. 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

Describes a class. The signature can include parentheses with parameters which will be shown as the constructor arguments.

attribute

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

Describes an object method. The parameters should not include the `self` parameter. The description should include similar information to that described for `function`.

opcode

Describes a Python *bytecode* instruction.

cmdoption

Describes a Python command line option or switch. Option argument names should be enclosed in angle brackets. Example:

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

```
    Run a module as a script.
```

envvar

Describes an environment variable that Python uses or defines.

There is also a generic version of these directives:

describe

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.
```

4.4 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 handled in a smart way:

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

```
.. highlightlang:: c
```

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

- The values normally used for the highlighting language are:

- python (the default)
- c
- rest
- none (no highlighting)

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

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. Documentation-specific include files should be placed in the `Doc/includes` subdirectory.

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

4.5 Inline markup

As said before, Sphinx uses interpreted text roles to insert semantic markup in documents.

Names of local variables, such as function/method arguments, are an exception, they should be marked simply with `*var*`.

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

There are some additional facilities 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.

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 should not include trailing parentheses to enhance readability. The parentheses are stripped when searching for identifiers.

data

The name of a module-level variable or constant.

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. 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.

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 the `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.

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

The following role does possibly create a cross-reference, but does not refer to objects:

TOKEN

The name of a grammar token (used in the reference manual to create links between production displays).

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, in fact, by default 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.

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.)

ENVVAR

An environment variable. 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``.

keyword

The name of a keyword in Python.

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.

option

A command-line option of Python. The leading hyphen(s) must be included. If a matching `cmdoption` directive exists, it is linked to. For options of other programs or scripts, use simple ```code``` markup.

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.

var

A Python or C variable or parameter name.

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.

4.6 Cross-linking markup

To support cross-referencing to arbitrary sections in the documentation, the standard reST labels are “abused” a bit: Every label must precede a section title; and every label name must be unique throughout the entire documentation source.

You can then reference to these sections using the `:ref: `label-name`` role.

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:` invocation is replaced with the section title.

4.7 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 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. In the interest of not scaring users away from pages filled with warnings, this directive should only be chosen over `note` for information regarding the possibility of crashes, data loss, or security implications.

versionadded

This directive documents the version of Python 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

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

impl-detail

This directive is used to mark CPython-specific information. Use either with a block content or a single sentence as an argument, i.e. either

```
.. impl-detail::

   This describes some implementation detail.

   More explanation.
```

or

```
.. impl-detail:: This shortly mentions an implementation detail.
```

“CPython implementation detail:” is automatically prepended to the content.

seealso

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

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

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

```
.. seealso::

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

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

rubric

This directive creates a paragraph heading that is not used to create a table of contents node. It is currently used for the “Footnotes” caption.

centered

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

```
.. centered::

   Paragraph contents.
```

4.8 Table-of-contents markup

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 files given in the directive body. A numeric `maxdepth` option may be given to indicate the depth of the tree; by default, all levels are included.

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

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

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

This accomplishes two things:

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

In the end, all files included in the build process must occur in one `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.

The special file `contents.rst` at the root of the source directory is the “root” of the TOC tree hierarchy; from it the “Contents” page is generated.

4.9 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.

The directive is `index` and 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
```

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).

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.

4.10 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. `unaryneg ::= "-" \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`
```

4.11 Substitutions

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

|release|

Replaced by the Python release the documentation refers to. This is the full version string including alpha/beta/release candidate tags, e.g. 2.5.2b3.

|version|

Replaced by the Python version the documentation refers to. This consists only of the major and minor version parts, e.g. 2.5, even for version 2.5.1.

|today|

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

DIFFERENCES TO THE LATEX MARKUP

Though the markup language is different, most of the concepts and markup types of the old LaTeX docs have been kept – environments as reST directives, inline commands as reST roles and so forth.

However, there are some differences in the way these work, partly due to the differences in the markup languages, partly due to improvements in Sphinx. This section lists these differences, in order to give those familiar with the old format a quick overview of what they might run into.

5.1 Inline markup

These changes have been made to inline markup:

- **Cross-reference roles**

Most of the following semantic roles existed previously as inline commands, but didn't do anything except formatting the content as code. Now, they cross-reference to known targets (some names have also been shortened): *mod* (previously *refmodule* or *module*)

func (previously *function*)

data (new)

const

class

meth (previously *method*)

attr (previously *member*)

exc (previously *exception*)

cdata

cfunc (previously *cfunction*)

cmacro (previously *csimplemacro*)

ctype

Also different is the handling of *func* and *meth*: while previously parentheses were added to the callable name (like `\func{str() }`), they are now appended by the build system – appending them in the source will result in double parentheses. This also means that `:func: `str(object) `` will not work as expected – use ``str(object) ``` instead!

- **Inline commands implemented as directives**

These were inline commands in LaTeX, but are now directives in reST: *deprecated*

versionadded

versionchanged

These are used like so:

```
.. deprecated:: 2.5
   Reason of deprecation.
```

Also, no period is appended to the text for *versionadded* and *versionchanged*. *note warning*

These are used like so:

```
.. note::

   Content of note.
```

- **Otherwise changed commands**

The *samp* command previously formatted code and added quotation marks around it. The *samp* role, however, features a new highlighting system just like *file* does:

```
:samp: `open({filename}, {mode})` results in `open(filename, mode)`
```

- **Dropped commands**

These were commands in LaTeX, but are not available as roles: *bfcode*

character (use `'''c' '''`)

citetitle (use ``Title <URL>`_`)

code (use ```code```)

email (just write the address in body text)

filenq

filevar (use the `{...}` highlighting feature of *file*)

programopt, *longprogramopt* (use *option*)

ulink (use ``Title <URL>`_`)

url (just write the URL in body text)

var (use `*var*`)

infinity, *plusminus* (use the Unicode character)

shortversion, *version* (use the `|version|` and `|release|` substitutions)

emph, *strong* (use the reST markup)

- **Backslash escaping**

In reST, a backslash must be escaped in normal text, and in the content of roles. However, in code literals and literal blocks, it must not be escaped. Example: `:file: 'C:\\Temp\\my.tmp'` vs. ```open("C:\\Temp\\my.tmp")```.

5.2 Information units

Information units (...*desc* environments) have been made reST directives. These changes to information units should be noted:

- **New names**

“desc” has been removed from every name. Additionally, these directives have new names:

cfuction (previously *cfuncdesc*)

cmacro (previously *csimplemacrodesc*)

exception (previously *excdesc*)

function (previously *funcdesc*)

attribute (previously *memberdesc*)

The *classdesc** and *excclassdesc* environments have been dropped, the *class* and *exception* directives support classes documented with and without constructor arguments.

- **Multiple objects**

The equivalent of the *...line* commands is:

```
.. function:: do_foo(bar)
                do_bar(baz)
```

Description of the functions.

IOW, just give one signatures per line, at the same indentation level.

- **Arguments**

There is no *optional* command. Just give function signatures like they should appear in the output:

```
.. function:: open(filename[, mode[, buffering]])
```

Description.

Note: markup in the signature is not supported.

- **Indexing**

The *...descni* environments have been dropped. To mark an information unit as unsuitable for index entry generation, use the *noindex* option like so:

```
.. function:: foo_*
   :noindex:
```

Description.

- **New information units**

There are new generic information units: One is called “describe” and can be used to document things that are not covered by the other units:

```
.. describe:: a == b
```

The equals operator.

The others are:

```
.. cmdoption:: -O
```

Describes a command-line option.

```
.. envvar:: PYTHONINSPECT
```

Describes an environment variable.

5.3 Structure

The LaTeX docs were split in several toplevel manuals. Now, all files are part of the same documentation tree, as indicated by the *toctree* directives in the sources (though individual output formats may choose to split them up into parts again). Every *toctree* directive embeds other files as subdocuments of the current file (this structure is not necessarily mirrored in the filesystem layout). The toplevel file is `contents.rst`.

However, most of the old directory structure has been kept, with the directories renamed as follows:

- `api` -> `c-api`
- `dist` -> `distutils`, with the single TeX file split up
- `doc` -> `documenting`
- `ext` -> `extending`
- `inst` -> `installing`
- `lib` -> `library`
- `mac` -> merged into `library`, with `mac/using.tex` moved to `using/mac.rst`
- `ref` -> `reference`
- `tut` -> `tutorial`, with the single TeX file split up

BUILDING THE DOCUMENTATION

You need to have Python 2.4 or higher installed; the toolset used to build the docs is written in Python. It is called *Sphinx*, it is not included in this tree, but maintained separately. Also needed are the docutils, supplying the base markup that Sphinx uses, Jinja, a templating engine, and optionally Pygments, a code highlighter.

6.1 Using make

Luckily, a Makefile has been prepared so that on Unix, provided you have installed Python and Subversion, you can just run

```
make html
```

to check out the necessary toolset in the *tools/* subdirectory and build the HTML output files. To view the generated HTML, point your favorite browser at the top-level index *build/html/index.html* after running “make”.

Available make targets are:

- “html”, which builds standalone HTML files for offline viewing.
- “htmlhelp”, which builds HTML files and a HTML Help project file usable to convert them into a single Compiled HTML (.chm) file – these are popular under Microsoft Windows, but very handy on every platform.
To create the CHM file, you need to run the Microsoft HTML Help Workshop over the generated project (.hlp) file.
- “latex”, which builds LaTeX source files as input to “pdflatex” to produce PDF documents.
- “text”, which builds a plain text file for each source file.
- “linkcheck”, which checks all external references to see whether they are broken, redirected or malformed, and outputs this information to stdout as well as a plain-text (.txt) file.
- “changes”, which builds an overview over all versionadded/versionchanged/ deprecated items in the current version. This is meant as a help for the writer of the “What’s New” document.
- “coverage”, which builds a coverage overview for standard library modules and C API.
- “pydoc-topics”, which builds a Python module containing a dictionary with plain text documentation for the labels defined in *tools/sphinxext/pyspecific.py* – pydoc needs these to show topic and keyword help.

A “make update” updates the Subversion checkouts in *tools/*.

6.2 Without make

You'll need to install the Sphinx package, either by checking it out via

```
svn co http://svn.python.org/projects/external/Sphinx-0.6.5/sphinx tools/sphinx
```

or by installing it from PyPI.

Then, you need to install Docutils, either by checking it out via

```
svn co http://svn.python.org/projects/external/docutils-0.6/docutils tools/docutils
```

or by installing it from <http://docutils.sf.net/>.

You also need Jinja2, either by checking it out via

```
svn co http://svn.python.org/projects/external/Jinja-2.3.1/jinja2 tools/jinja2
```

or by installing it from PyPI.

You can optionally also install Pygments, either as a checkout via

```
svn co http://svn.python.org/projects/external/Pygments-1.3.1/pygments tools/pygments
```

or from PyPI at <http://pypi.python.org/pypi/Pygments>.

Then, make an output directory, e.g. under *build/*, and run

```
python tools/sphinx-build.py -b<builder> . build/<outputdirectory>
```

where *<builder>* is one of *html*, *text*, *latex*, or *htmlhelp* (for explanations see the make targets above).

GLOSSARY

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

. . . The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See *2to3 - Automated Python 2 to 3 code translation* (in *The Python Library Reference*).

abstract base class Abstract Base Classes (abbreviated ABCs) complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABC with the `abc` module.

argument A value passed to a function or method, assigned to a named local variable in the function body. A function or method may have both positional arguments and keyword arguments in its definition. Positional and keyword arguments may be variable-length: `*` accepts or passes (if in the function definition or call) several positional arguments in a list, while `**` does the same for keyword arguments in a dictionary.

Any expression may be used within the argument list, and the evaluated value is passed to the local variable.

attribute A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

BDFL Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode.

A list of bytecode instructions can be found in the documentation for *the dis module* (in *The Python Library Reference*).

class A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes will be removed in Python 3.0.

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3,

but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See [PEP 343](#).

CPython The canonical implementation of the Python programming language. The term “CPython” is used in contexts when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

See the *documentation for function definition* (in *The Python Language Reference*) for more about decorators.

descriptor Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see *Implementing Descriptors* (in *The Python Language Reference*).

dictionary An associative array, where arbitrary keys are mapped to values. The use of `dict` closely resembles that for `list`, but the keys can be any object with a `__hash__()` function, not just integers. Called a hash in Perl.

docstring A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing A pythonic programming style which determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object (“If it looks like a duck and quacks like a duck, it must be a duck.”) By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

extension module A module written in C or C++, using Python's C API to interact with the core and with user code.

finder An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See [PEP 302](#) for details.

function A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also *argument* and *method*.

`__future__` A pseudo module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to 2. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to 2.75. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator A function which returns an iterator. It looks like a normal function except that values are returned to the caller using a `yield` statement instead of a `return` statement. Generator functions often contain one or more `for` or `while` loops which `yield` elements back to the caller. The function execution is stopped at the `yield` keyword (returning the result) and is resumed there when the next element is requested by calling the `next()` method of the returned iterator.

generator expression An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

GIL See *global interpreter lock*.

global interpreter lock The lock used by Python threads to assure that only one thread executes in the *CPython virtual machine* at a time. This simplifies the CPython implementation by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. Efforts have been made in the past to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity), but so far none have been successful because performance suffered in the common single-processor case.

hashable An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their `id()`.

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python. Good for beginners, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.

immutable An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also `__future__`.

importer An object that both finds and loads a module; both a *finder* and *loader* object.

interactive Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

iterable A container object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in *Iterator Types* (in *The Python Library Reference*).

keyword argument Arguments which are preceded with a `variable_name=` in the call. The variable name designates the local name in the function to which the value is assigned. `**` is used to accept or pass a dictionary of keyword arguments. See *argument*.

lambda An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [arguments]: expression`

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

list A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are $O(1)$.

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

mapping A container object (such as `dict`) which supports arbitrary key lookups using the special method `__getitem__()`.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in *Customizing class creation* (in *The Python Language Reference*).

method A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

mutable Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattr__()`.

More information can be found in *New-style and classic classes* (in *The Python Language Reference*).

object Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

positional argument The arguments assigned to local names inside a function or method, determined by the order in which they were given in the call. * is used to either accept multiple positional arguments (when in the definition), or pass several arguments as a list to a function. See *argument*.

Python 3000 Nickname for the next major Python version, 3.0 (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

Pythonic An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):  
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:  
    print piece
```

reference count The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

special method A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in *Special method names* (in *The Python Language Reference*).

statement A statement is part of a suite (a “block” of code). A statement is either an *expression* or a one of several constructs with a keyword, such as `if`, `while` or `print`.

triple-quoted string A string which is bound by three instances of either a quotation mark (“) or an apostrophe (‘). While they don’t provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type The type of a Python object determines what kind of object it is; every object has a type. An object’s type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

view The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They are lazy sequences that will see changes in the underlying dictionary. To force

the dictionary view to become a full list use `list(dictview)`. See *Dictionary view objects* (in *The Python Library Reference*).

virtual machine A computer defined entirely in software. Python’s virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “`import this`” at the interactive prompt.

ABOUT THESE DOCUMENTS

These documents are generated from [reStructuredText](#) sources by [Sphinx](#), a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain takes place on the docs@python.org mailing list. We're always looking for volunteers wanting to help with the docs, so feel free to send a mail there!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the [Docutils](#) project for creating [reStructuredText](#) and the [Docutils](#) suite;
- Fredrik Lundh for his [Alternative Python Reference](#) project from which [Sphinx](#) got many good ideas.

See [Reporting Bugs](#) for information how to report bugs in this documentation, or Python itself.

B.1 Contributors to the Python Documentation

This section lists people who have contributed in some way to the Python documentation. It is probably not complete – if you feel that you or anyone else should be on this list, please let us know (send email to docs@python.org), and we'll be glad to correct the problem.

Aahz, Michael Abbott, Steve Alexander, Jim Ahlstrom, Fred Allen, A. Amoroso, Pehr Anderson, Oliver Andrich, Heidi Annexstad, Jesús Cea Avi3n, Manuel Balsera, Daniel Barclay, Chris Barker, Don Bashford, Anthony Baxter, Alexander Belopolsky, Bennett Benson, Jonathan Black, Robin Boerdijk, Michal Bozon, Aaron Brancotti, Georg Brandl, Keith Briggs, Ian Bruntlett, Lee Busby, Lorenzo M. Catucci, Carl Cerecke, Mauro Cicognini, Gilles Civario, Mike Clarkson, Steve Clift, Dave Cole, Matthew Cowles, Jeremy Craven, Andrew Dalke, Ben Darnell, L. Peter Deutsch, Robert Donohue, Fred L. Drake, Jr., Josip Dzolong, Jeff Epler, Michael Ernst, Blame Andy Eskilsson, Carey Evans, Martijn Faassen, Carl Feynman, Dan Finnie, Hern3n Mart3nez Foffani, Stefan Franke, Jim Fulton, Peter Funk, Lele Gaifax, Matthew Gallagher, Gabriel Genellina, Ben Gertzfield, Nadim Ghaznavi, Jonathan Giddy, Shelley Gooch, Nathaniel Gray, Grant Griffin, Thomas Guettler, Anders Hammarquist, Mark Hammond, Harald Hanche-Olsen, Manus Hand, Gerhard H3ring, Travis B. Hartwell, Tim Hatch, Janko Hauser, Thomas Heller, Bernhard Herzog, Magnus L. Hetland, Konrad Hinsen, Stefan Hoffmeister, Albert Hofkamp, Gregor HOFFleit, Steve Holden, Thomas Holenstein, Gerrit Holl, Rob Hooft, Brian Hooper, Randall Hopper, Michael Hudson, Eric Huss, Jeremy Hylton, Roger Irwin, Jack Jansen, Philip H. Jensen, Pedro Diaz Jimenez, Kent Johnson, Lucas de Jonge, Andreas Jung, Robert Kern, Jim Kerr, Jan Kim, Greg Kochanski, Guido Kollerie, Peter A. Koren, Daniel Kozan, Andrew M. Kuchling, Dave Kuhlman, Erno Kuusela, Thomas Lamb, Detlef Lannert, Piers Lauder, Glyph Lefkowitz, Robert Lehmann, Marc-Andr3 Lemburg, Ross Light, Ulf A. Lindgren, Everett Lipman, Mirko Liss, Martin von L3wis, Fredrik Lundh, Jeff MacDonald, John Machin, Andrew MacIntyre, Vladimir Marangozov, Vincent Marchetti, Laura Matson, Daniel May, Rebecca McCreary, Doug Mennella, Paolo Milani, Skip Montanaro, Paul Moore, Ross Moore, Sjoerd Mullender, Dale Nagata, Ng Pheng Siong, Koray Oner, Tomas Ooppelstrup, Denis S. Otkidach, Zooko O'Whielacronx, Shriphani Palakodety, William Park, Joonas Paalasmaa, Harri Pasanen, Bo Peng, Tim Peters, Benjamin Peterson, Christopher

Petrilli, Justin D. Pettit, Chris Phoenix, François Pinard, Paul Prescod, Eric S. Raymond, Edward K. Ream, Terry J. Reedy, Sean Reifschneider, Bernhard Reiter, Armin Rigo, Wes Rishel, Armin Ronacher, Jim Roskind, Guido van Rossum, Donald Wallace Rouse II, Mark Russell, Nick Russo, Chris Ryland, Constantina S., Hugh Sasse, Bob Savage, Scott Schram, Neil Schemenauer, Barry Scott, Joakim Sernbrant, Justin Sheehy, Charlie Shepherd, Michael Simcich, Ionel Simionescu, Michael Sloan, Gregory P. Smith, Roy Smith, Clay Spence, Nicholas Spies, Tage Stabell-Kulo, Frank Stajano, Anthony Starks, Greg Stein, Peter Stoehr, Mark Summerfield, Reuben Sumner, Kalle Svensson, Jim Tittsler, David Turner, Ville Vainio, Martijn Vries, Charles G. Waldman, Greg Ward, Barry Warsaw, Corran Webster, Glyn Webster, Bob Weiner, Eddy Welbourne, Jeff Wheeler, Mats Wichmann, Gerry Wiener, Timothy Wild, Paul Winkler, Collin Winter, Blake Winton, Dan Wolfe, Steven Work, Thomas Wouters, Ka-Ping Yee, Rory Yorke, Moshe Zadka, Milan Zamazal, Cheng Zhang.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

HISTORY AND LICENSE

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <http://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <http://www.opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes

Continued on next page

Table C.1 – continued from previous page

2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.4.4	2.4.3	2006	PSF	yes
2.5	2.4	2006	PSF	yes
2.5.1	2.5	2007	PSF	yes
2.5.2	2.5.1	2008	PSF	yes
2.5.3	2.5.2	2008	PSF	yes
2.6	2.5	2008	PSF	yes
2.6.1	2.6	2008	PSF	yes
2.6.2	2.6.1	2009	PSF	yes
2.6.3	2.6.2	2009	PSF	yes
2.6.4	2.6.3	2010	PSF	yes
2.7	2.6	2010	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.7

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.7 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.7 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2010 Python Software Foundation; All Rights Reserved" are retained in Python 2.7 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.7 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.7.
4. PSF is making Python 2.7 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.7 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

- By copying, installing or otherwise using Python 2.7, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

- This LICENSE AGREEMENT is between BeOpen.com (“BeOpen”), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (“Licensee”) accessing and otherwise using this software in source or binary form and its associated documentation (“the Software”).
- Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
- BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
- BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
- This License Agreement will automatically terminate upon a material breach of its terms and conditions.
- This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
- By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

- This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
- Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>.”
- In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from <http://www.math.keio.ac.jp/matumoto/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
<http://www.math.keio.ac.jp/matsumoto/emt.html>
email: matsumoto@math.keio.ac.jp

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate
source files from the WIDE Project, <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.


```
| reflect those of the United States Government or the University |
| of California, and shall not be used for advertising or product |
\ endorsement purposes. /
```

C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.
```

```
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
L. Peter Deutsch
ghost@aladdin.com
```

```
Independent implementation of MD5 (RFC 1321).
```

```
This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
```

```
http://www.ietf.org/rfc/rfc1321.txt
```

```
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A. It does not include
any code or documentation that is identified in the RFC as being
copyrighted.
```

```
The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>. Other authors are noted in the change history
that follows (in reverse chronological order):
```

```
2002-04-13 lpd Removed support for non-ANSI compilers; removed
references to Ghostscript; clarified derivation from RFC 1321;
now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
added conditionalization for C++ compilation from Martin
Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.6 Cookie management

The `Cookie` module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 Execution tracing

The trace module contains the following notice:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
<http://zooko.com/>
<mailto:zooko@zooko.com>

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies,

and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

C.3.9 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
```

```
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Modified by Jack Jansen, CWI, July 1995:

- Use `binascii` module to do the actual line-by-line conversion between `ascii` and `binary`. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.10 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
The XML-RPC client interface is
```

```
Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh
```

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

```
Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
```

prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.11 test_epoll

The test_epoll contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.12 Select kqueue

The select and contains the following notice for the kqueue interface:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND

ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.13 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <http://www.netlib.org/fp/>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/*
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 */
```

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2010 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See *History and License* for complete license and permissions information.

INDEX

Symbols

..., 31
__future__, 33
__slots__, 36
>>>, 31
2to3, 31

A

abstract base class, 31
argument, 31
attribute, 31

B

BDFL, 31
bytecode, 31

C

class, 31
classic class, 31
coercion, 31
complex number, 32
context manager, 32
CPython, 32

D

decorator, 32
descriptor, 32
dictionary, 32
docstring, 32
duck-typing, 32

E

EAFP, 32
expression, 33
extension module, 33

F

finder, 33
function, 33

G

garbage collection, 33
generator, 33
generator expression, 33
GIL, 33
global interpreter lock, 33

H

hashable, 33

I

IDLE, 34
immutable, 34
importer, 34
integer division, 34
interactive, 34
interpreted, 34
iterable, 34
iterator, 34

K

keyword argument, 34

L

lambda, 34
LBYL, 35
list, 35
list comprehension, 35
loader, 35

M

mapping, 35
metaclass, 35
method, 35
mutable, 35

N

named tuple, 35
namespace, 35
nested scope, 35
new-style class, 35

O

object, 35

P

positional argument, 36

Python 3000, 36

Python Enhancement Proposals

 PEP 302, 33, 35

 PEP 343, 32

Pythonic, 36

R

reference count, 36

S

sequence, 36

slice, 36

special method, 36

statement, 36

T

triple-quoted string, 36

type, 36

V

view, 36

virtual machine, 37

Z

Zen of Python, 37