sphinxcontrib-emacs

Release 0.4a0

January 01, 2015

Contents

sphinxcontrib-emacs is a Sphinx extension to document Emacs code.

CHAPTER 1

Features

- Description directives for Emacs Lisp symbols
- Autodoc for Emacs Lisp code
- Better Texinfo integration to build online manuals for Emacs

CHAPTER 2

User Guide

The user guide documents how to use sphinxcontrib-emacs to document your Emacs projects. It starts with an introduction, and a quickstart guide, and then goes on to document each feature of sphinxcontrib-emacs.

2.1 Introduction

sphinxcontrib-emacs is an extension to the Sphinx documentation tool, which lets you document Emacs Lisp symbols, including automatic docstring extraction from Emacs Lisp source.

This page gives you brief overview of Sphinx and its underlying markup reStructuredText.

2.1.1 Sphinx

Sphinx forms the base this extension builds on. It is a powerful documentation processor, initially created by the Python Foundation for the Python documentation.

It supports many different output formats (including HTML, PDF and Texinfo), has extensive cross-referencing support, automated index generation, and a powerful interface for custom extensions.

More information about Sphinx can be found in the Sphinx documentation. If you are not familiar with Sphinx, please take a look at the Introduction, and read the tutorial.

2.1.2 reStructuredText

Sphinx uses reStructuredText (REST (reStructuredText)) as markup for documentation sources. It is an easy-to-use plaintext markup, similar to the popular Markdown format, but with a more regular grammar, and extensible syntax.

The Sphinx documentation has a brief, but comprehensive *reStructuredText Primer*, which should get you started on reST if you are familiar with similar formats like Markdown. More information can be found on the reStructuredText page, including a specification and a reference of the provided directives and roles.

Sphinx adds its own special markup to reStructuredText, for the specific needs of large documentation. This markup is documented in *Sphinx Markup Constructs*.

2.2 Installation

This document covers the installation and setup of this extension.

2.2.1 Installing the package

As sphinxcontrib-emacs is available in Python's package repository, you can easily install it with pip, Python's package manager:

```
$ pip install sphinxcontrib-emacs
```

2.2.2 Enabling the extension

To enable this extension, simply add it to `extensions` in your `conf.py`:

```
extensions = ['sphinx.ext.intersphinx',
              'sphinxcontrib.emacs']
```

2.3 Quickstart by example

This page gets you started quickly with sphinxcontrib-emacs, by an simple example. It assumes

- that you are already familiar with Sphinx,
- and have Sphinx and sphinxcontrib-emacs installed.

Thus, this page will only go through the specific setup and features of sphinxcontrib-emacs. Consult the Sphinx documentation for information about Sphinx.

> **Warning:** If you are not familiar with Sphinx, please go through the Sphinx tutorial first.

2.3.1 The project layout

We want to document a simple `hello.el` library, which implements the good old Hello world program.

Our project has the following layout:

```
README.md
hello.el
```

`hello.el` is very simple, with just one command, and a user option:

```
(defcustom greeting "Hello %s!"
  "The greeting to use in `greet'.

The value of this variable should contain is used as string for
`format', with the only argument being the name of the user to
greet."
  :safe #'stringp
  :package-version '(hello . "0.1"))

(make-variable-buffer-local 'greeting)

(defun greet (name)
  "Greet the user with the given NAME, in classic Hello world style.

When called interactively, prompt for NAME.

```

```
Use the message template from 'greeting' to assemble the greeting
mesage.

See URL 'http://en.wikipedia.org/wiki/Hello_world_program'."
  (interactive "sYour name: ")
  (message greeting name))

(global-set-key (kbd "C-c g") #'greet)
```

2.3.2 Setup

We create a subdirectory `doc/` for our documentation, and go through the **sphinx-quickstart** tool, which gives us a skeleton for our documentation:

```
doc/conf.py
doc/index.rst
doc/Makefile
doc/make.bat
```

`doc/conf.py` is the *build configuration*. Here we enable sphinxcontrib-emacs by adding it to the `extensions` setting:

```
# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = ['sphinxcontrib.emacs']
```

2.3.3 Documenting commands

We start with documenting the interactive command, the most important feature of our `hello` library.

We create a separate document `doc/hello.rst` for the documentation of the `hello` library, and add it to the *TOC tree* in `doc/index.rst`:

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

   hello
```

In `doc/hello.rst` we can now document the `hello` command:

```
==================
 The hello library
==================

Greet users.

.. el:command:: greet
   :binding: C-c g

   Prompt for the name of a user and greet them.

   Use :el:option:`greeting` to change the greeting text.
```

`:el:option:`greeting`` inserts a cross-reference to the documentation of the `greeting` option, which we add *further down*.

In `hello.el`, we install global key binding `C-c g` for our command, so we document this additional binding in the `:binding:` option. This binding now appears in the documentation of the `greet` command:

C-c g
M-x greet
Prompt for the name of a user and greet them.

Use `greeting` to change the greeting text.

2.3.4 Documenting different invocations of commands

In Emacs Lisp, any command is a function as well, which can be called from Emacs Lisp. To account for this, we also want to document the function "aspect" of `greet`:

```
.. el:function:: greet name
   :noindex:

   Greet the user with the given ``name``.

   ``name`` is a string identifying the user to greet.
```

Function greet (name)
Greet the user with the given `name`.

`name` is a string identifying the user to greet.

We use the `:noindex:` option to suppress the index entry and cross-reference target, since we have already documented the `greet` symbol as command.

In the documentation index, and in cross-references, `greet` will always refer to the interactive command, documented above.

2.3.5 Auto-documenting functions

When documenting `greet` as a function, we wrote an explicit documentation. However, we already have documentation, in the docstring of `greet`:

```
(defun greet (name)
  "Greet the user with the given NAME, in classic Hello world style.

When called interactively, prompt for NAME.

Use the message template from 'greeting' to assemble the greeting
mesage.

See URL 'http://en.wikipedia.org/wiki/Hello_world_program'."
  (interactive "sYour name: ")
  (message greeting name))
```

To avoid this duplication, we change our documentation to extract the docstring of the `greet` function.

First, we need to point sphinxcontrib-emacs to the Emacs Lisp source file, by changing appending the following to `doc/conf.py`:

```
emacs_lisp_load_path = [
    os.path.abspath(os.path.join(os.path.dirname(__file__), os.pardir))
]
```

This tells sphinxcontrib-emacs, that Emacs Lisp source files are to be found in the parent directory of the documentation (remember our *project layout*). By using `os.path.join()` and `os.pardir`, we ensure that our documentation builds regardless of where we checked out our project.

Then we load the `hello` library to make its docstrings available, by adding the following to the top of `doc/hello.rst`:

Now we use the docstring of `greet` as documentation of the `greet` function:

```
.. el:require:: hello

.. el:function:: greet
   :noindex:
   :auto:
```

Function `greet` (name)

```
Greet the user with the given NAME, in classic Hello world style.

When called interactively, prompt for NAME.

Use the message template from 'greeting' to assemble the greeting
mesage.

See URL 'http://en.wikipedia.org/wiki/Hello_world_program'.
```

We remove our manually written documentation, and instead add the `:auto:` option to have docstring of `greet` inserted as documentation. Note that we also removed the argument name, since `:auto:` will also extract the function signature.

In docstrings, only the markup of Help Mode is parsed (see *Documentation Tips(elisp)*). reST is not supported.

Note: The peculiar layout and presentation of extracted docstrings is to maintain compatibility with the limited markup and presentation of docstrings in Emacs' Help Mode.

2.3.6 Auto-documenting options

Now we can also automatically document the `greeting` option:

```
.. el:option:: greeting
   :auto:
```

Using `:auto:` for variables will automatically add the variables properties as well. Remember the definition of `greeting`:

```
(defcustom greeting "Hello %s!"
  "The greeting to use in 'greet'.

The value of this variable should contain is used as string for
'format', with the only argument being the name of the user to
greet."
  :safe #'stringp
  :package-version '(hello . "0.1"))

(make-variable-buffer-local 'greeting)
```

The variable is marked as safe for string values, and as buffer-local. These properties are reflected in the documentation. The package version is recorded as well:

User Option `greeting`

Variable properties

Automatically becomes buffer-local when set. This variable is safe as a file local variable if its value satisfies the predicate `stringp`.

```
The greeting to use in 'greet'.

The value of this variable should contain is used as string for
'format', with the only argument being the name of the user to
greet.
```

This user option was introduced, or its default value was changed, in version 0.1 of the hello package.

2.3.7 Further reading

- *Documenting Emacs Lisp symbols* describes directives and roles to document Emacs Lisp symbols.
- *Extracting docstrings of Emacs Lisp symbols* has details on auto-documenting Emacs Lisp symbols.

2.4 Documenting Emacs Lisp symbols

This page introduces the Emacs Lisp domain and its directives and roles to document and cross-reference Emacs Lisp symbols.

Please read *Sphinx Domains* in the Sphinx documentation for more information on the concept of domains.

2.4.1 The Emacs Lisp domain

These directives live in the `el` domain.

You can set this domain as default domain to refer to the directives and roles without the `el:` prefix. To change the default domain globally, set the configuration value `primary_domain` in your `conf.py` accordingly:

```
primary_domain = 'el'
```

To change the default domain per file, use the `default-domain` directive:

```
.. default-domain:: el
```

2.4.2 Common options

All directives support some common options:

`:noindex:` Suppress the creation of an index entry. See *Documenting interactive commands* for an example of using this directive.

`:auto:` Insert the docstring extracted from the Emacs Lisp source code of this symbol.

For function-like symbols, also use the extracted function signature. For variables, also insert various various variable properties, such as whether the variable is safe as buffer local variable, whether it is automatically buffer local, etc.

Generally, the output of this option closely mirrors the appearance of docstrings in Emacs' built-in Help Mode.

See *Extracting docstrings of Emacs Lisp symbols* for more information about automatically extracting docstrings.

2.4.3 Scopes

An Emacs Lisp symbol can be defined in different "scopes":

```
(defvar hello "Hello")

(defun hello (name)
  (message "%s %s" hello name))
```

This defines `hello` as variable *and* as function. You can document these two definitions independently with the appropriate directives:

```
.. el:defvar:: hello

.. el:function:: hello
```

2.4.4 Documenting functions and macros

`.. el:function:: symbol [argument ...] [&optional optional ...] [&rest args]`
`.. el:macro:: symbol [argument ...] [&optional optional ...] [&rest args]`

Document `symbol` as function or macro with the given arglist, for example:

```
(defun hello (name &optional greeting)
  (message "%s %s" (or greeting "Hello") name))

.. el:function:: hello name &optional greeting

   Greet the user with the given ``name``.

   If ``greeting`` is given, use it as greeting, instead of the standard
   "Hello".
```

Use `el:function` and `el:macro` to cross-reference symbols described with these directives.

`:el:function:`
`:el:macro:`

Add a reference to a function or macro.

2.4.5 Documenting interactive commands

`.. el:command:: symbol`

Document `symbol` as interactive command:

```
(defun greet (name)
  (interactive "%M")
  (message "Hello %s" name))

.. el:command:: greet
   :binding: C-c g

   Prompt for a name and greet the user.
```

Commands are described as a user would type them in Emacs, via `M-x`, and optionally by specific bindings. Hence, the above example would look like this:

C-c g
M-x greet
Prompt for a name and greet the user.

`prefix-arg` adds the given prefix argument to the keybindings:

```
.. el:command:: greet
   :binding: C-c g
   :prefix-arg: C-u

   Greet the current user.
```

C-u C-c g
C-u M-x greet
Greet the current user.

:el:command:
Reference an Emacs Lisp command.

Since commands are just functions, this directive is the same as `el:function`.

Documenting different invocations of a command

Emacs Lisp commands can be invoked in different ways, e.g. with or without prefix arguments, with different prefix arguments, or as ordinary function from Emacs Lisp.

This extension encourages you to document all variants of a command *independently*:

```
.. el:command:: greet
   :binding: C-c g

   Prompt for a name and greet the user.

.. el:command:: greet
   :binding: C-c g
   :prefix-arg: C-u
   :noindex:

.. el:function:: greet name
   :noindex:

   Show a greeting message for the user with the given ``name``.
```

This example documents three different variants of the Emacs Lisp command `greet`: Without prefix argument, with universal prefix argument, and as Emacs Lisp function.

To avoid ambiguities in the index and when resolving cross-references, you must add the `noindex` option to all but the most "important" variant of the command.

In the above example, we presume that `C-c g` is the most important variant, so we add `:noindex:` to all others. The index entry and cross-references with thus point to the `C-c g` variant.

2.4.6 Documenting constants, variables, user options and hooks

```
.. el:constant:: symbol
.. el:variable:: symbol
```

.. **el:option::** symbol
.. **el:hook::** symbol
Document symbol as (constant) Emacs Lisp variable, for example:

```
(defvar python-check-command "pylint")

.. el:variable:: python-check-command
   :local:
   :safe: stringp

   The shell command to use for checking the current buffer.
```

This documents python-check-command as buffer-local variable which is safe as local variable when its value matches the predicate stringp.

The flag :local: denotes that the variable is automatically buffer-local.

The option :safe: denotes that the variable is safe as local variable with the given predicate. If the predicate is a symbol, its function definition is cross-referenced.

The flag :risky: denotes that the variable is risky to use as local variable.

With el:option or el:hook, document symbol as customizable user option or hook respectively. This does not affect cross-referencing, but uses a different description text for symbol.

Use el:option, el:variable, or el:hook to cross-reference symbols described with these directives.

:el:variable:
:el:option:
:el:hook:
Insert a reference to a variable, option or hook respectively.

2.4.7 Documenting faces

.. **el:face::** symbol
Document symbol as a face, for example:

```
(defface error '((t :foreground red)))

.. el:face:: error

   The face for errors.
```

:el:face:
Insert a reference to a face.

2.4.8 Documenting CL structs

.. **el:cl-struct::** symbol
Document symbol as Cl struct defined by cl-defstruct:

```
(cl-defstruct (person
               (:constructor person-new)
               (:constructor person-with-name name))
  name mobile)
```

```
.. el:cl-struct:: person

   A person.

   .. el:cl-slot:: name

      The name of a person

   .. el:cl-slot:: mobile

      The mobile phone number

.. el:defun:: person-new :name name :mobile mobile

   Create a new person with the given ``name`` and ``mobile`` phone
   number.

.. el:defun:: person-with-name name

   Create a new person with the given ``name``.
```

Document constructors as standard functions with `el:function`. For slots, use the special `el:cl-slot` directive:

`.. el:cl-slot:: slot`
: Documents `slot` as a slot of the current Cl struct.

  Warning: Using this directive **outside** of a `el:cl-struct` block is an error.

  As Cl slots are functions in Emacs Lisp, this directive creates a function reference to the slot. Hence, the `name` slot from the above example can be referenced either with `el:cl-slot` or with `el:function`:

  ```
  The slot :el:cl-slot:`~person name` holds the name of a person.

  To get the name, call :el:function:`person-name`.
  ```

  In this example, both references would point to the description of `name` as in the example above. The difference is merely in presentation: While `el:function` always shows the entire function name, role:`el:cl-slot` only shows the name of the slot, if the reference appears inside a `el:cl-struct` block, or if the role text starts with a tilde.

`:el:cl-slot:`
: Reference a slot of a Cl structure.

  The text of the role has the form *`struct slot`* where `struct` is the name of the structure containing the given `slot`. Inside of a `el:cl-struct` block, `struct` may be omitted in which case it defaults to the current structure.

  When referencing a slot of the current structure inside a `el:cl-struct` block, the name of the struct is omitted in the output. To explicitly omit the struct name, prefix the role text with `~`, as in `:el:cl-slot:`~person name``.

2.5 Extracting docstrings of Emacs Lisp symbols

sphinxcontrib-emacs can extract docstrings from Emacs Lisp libraries, and use them in your documentation.

This page explains the advantages and drawbacks of this approach, and guides you through setup and usage.

Warning: When extracting docstrings from Emacs Lisp source code, be sure to respect the license of the corresponding source code, which covers the docstrings as well.

2.5.1 Pros and cons

Re-using docstrings for documentation avoids duplication and provides consistent information regardless of whether a user reads your documentation or the docstring right in Emacs. This comes at a price however, and there are two major drawbacks.

Limits of the interpreter

To avoid an external dependency onto `emacs`, sphinxcontrib-emacs uses a custom Emacs Lisp reader and interpreter to extract docstrings. This reader and interpreter is limited in what it supports:

- Static top-level definitions like `defun`, `defvar`, `defcustom` and friends.
- Top-level `put` and related forms (e.g. `make-variable-buffer-local`) with static arguments, that is, quoted symbols and literal primitives
- Body forms of `eval-and-compile`, `progn`, and related forms

Notably, it does **not** expand macros, inspect the body forms of definitions or track the values of variables.

While this limited reader and interpreter is usually sufficient to extract docstrings, since definitions tend to be static in Emacs Lisp, but it will obviously fail in specific cases:

- Nested definitions, e.g. a `defun` within a `defun`.
- Definitions in macro expansions, e.g. a `defvar` expanded by a custom macro.
- Manually assembled definitions, e.g. explicitly setting the function cell of a symbol and its `function-documentation` property.

In these cases, sphincontrib-emacs will fail to properly extract docstrings. As of now, there is no way to work around these limitations, other than writing documentation manually.

Primitive markup of docstrings

To ensure compatibility with the limited markup and presentation capabilities of Emacs' Help Mode, sphinxcontrib-emacs shows docstrings as literal block of text, and does not parse reST markup inside docstrings. Only references as documented in *Documentation Tips(elisp)* are handled.

Note: Key binding substitutions (see *Keys in Documentation(elisp)*) are **not** substituted, since the limited Emacs Lisp interpreter used by this extension does not track the values of keymaps (see *Limits of the interpreter*).

Thus, extracted docstrings look somewhat "ugly" compared to manually written documentation.

2.5.2 Loading the Emacs Lisp source

To use docstrings from Emacs Lisp source, you first need to load the corresponding Emacs Lisp source file with `el:require` from the *Emacs Lisp domain*:

.. **el:require:: feature**
Load the given `feature` to make its docstrings available for auto-documenting Emacs Lisp symbols. `feature` is a feature symbol, like in the Emacs function `require`.

sphinxcontrib-emacs searches for the corresponding source file in `emacs_lisp_load_path`, which is similar to Emacs' `load-path`.

Note: You must put `el:require` **before** the first auto-documented Emacs Lisp symbol in a file. Also, you must add the necessary `el:require` directives to **every** file which uses docstrings from an Emacs Lisp source.

`el:require` searches for source files in `emacs_lisp_load_path`:

emacs_lisp_load_path
A list of directories where to look for Emacs Lisp sources.

Set this in your `conf.py`, to point sphinxcontrib-emacs to the location of the Emacs Lisp source whose docstrings you want to use. For instance, if your Emacs Lisp library sits in the top-level source directory, and your `conf.py` in the subdirectory `doc/`, you would add the following to `conf.py`:

```
import os

emacs_lisp_load_path = [
    os.path.abspath(os.path.join(os.path.dirname(__file__), os.pardir))
]
```

2.5.3 Using docstrings

To insert the docstring of a symbol, add the `:auto:` flag to the corresponding directive:

Warning: Currently, `el:cl-struct` and `el:cl-slot` do not support `:auto:` properly.

With `auto`, all directives from the *Emacs Lisp domain* will

- insert the docstring of the symbol before any additional content of the directive,
- and add a `versionchanged` annotation if appropriate.

`el:function` will also extract the function signature from the Emacs Lisp source. Any custom signature is *ignored*.

Furthermore, `el:variable`, `el:option` and `el:hook` insert annotations concerning the properties of a variable:

- Whether the variable is buffer local or not.
- Whether the variable is safe or risky as a file-local variable.

2.6 Texinfo support

This page documents the special Texinfo support offered by this extension.

2.6.1 Introduction

Emacs users expect to have documentation available as Info manuals, to read and browse documentation with Emacs' Info viewer without having to leave Emacs. Sphinx provides a `Texinfo builder` to generate a Texinfo file, which can be feed to `makeinfo` 4.x and upwards. While the results are not as perfect as a manually written Texinfo manual, they are good enough for daily use.

2.6.2 Setup

To generate Texinfo documentation, you need tell the Texinfo builder which documents to include into the manual, by setting `texinfo_documents` in `conf.py`:

```
texinfo_documents = [('index', 'hello', '', 'John Doe',
                      'hello', 'Hello world for GNU Emacs', 'Emacs',
                      False)]
```

Assuming that `index.rst` is the root document, this will include the entire documentation in the Info manual. Typically, though, this is not what you want, since some parts of your documentation will be specific to HTML, such as a nice front page with a screenshot to "advertise" your project.

Hence, you may want to include only the actual manual in the Info manual:

```
texinfo_documents = [('manual', 'hello', '', 'John Doe',
                      'hello', 'Hello world for GNU Emacs', 'Emacs',
                      False)]
```

To fine-tune the Texinfo output, take a look at the other *Options for Texinfo output*.

2.6.3 Referencing Info manuals

To reference pages ("nodes") in other Info manuals, use the special `infonode` role:

:infonode:
Reference a node in an Info manual.

The text of this role has the form `(manual)node`, where `manual` is the name of the Info manual, and `node` the name of the target node in `manual`. For instance, ``:infonode:`(emacs)Intro` `` references the Introduction of the GNU Emacs manual: *Intro(emacs)*.

When generating Texinfo output, this role creates a special Info reference. For other output formats, this role creates a standard reference to the online version of the Info manual.

The online version of the Info manual is looked up in the latest Info manual database of Texinfo. Use `info_xref` to add explicit entries to the database.

info_xref
Additional Info manuals for cross-referencing.

The value is a dictionary, mapping the name of a manual to the **root URL** of the split HTML version, i.e. the HTML version that has a single page per node.

For instance, the following setting adds the manual of ERT, the Emacs unit testing library:

```
info_xref = {'ert': 'http://www.gnu.org/software/emacs/manual/html_node/ert/'}
```

Entries in this configuration value override entries retrieved from the online database. The online database is still consulted for other manuals, though.

In Info manuals these special references have a couple of advantages over a standard reference to the online version of the referenced manual:

- The Info reader of GNU Emacs can follow these references directly inside Emacs, without the need for a proper Web browser, and keeps a consistent navigation history across references. For instance, when following a reference to the Emacs manual, the user can press `L` in the Emacs manual to get back to the reference.
- The references work without a network connection, because Info manuals are stored on disk and can be read and browsed offline.

Hence, the experience of using your manual in Emacs is more consistent with these special references.

The downside is that you can only reference nodes in other manuals, but no entities within nodes, i.e. you can reference the *Rearrangement(elisp)* node in the Emacs Lisp reference, but not the documentation of the nreverse function in this node.

2.6.4 Special inline markup

The following roles let you denote metavariables, which get special rendering in Info manuals. They are typically used to refer to parameters of functions.

:var:
: Denote a metavariable.

  In HTML, the text of this role is enclosed in a `var` tag. In Texinfo, it is rendered using the `@var` macro.

:varcode:
: Like `samp`, but denote text in curly braces as metavariable (as in `var`) instead of emphasizing it.

CHAPTER 3

Contribute

This section of the documentation explains how to contribute to sphinxcontrib-emacs on Github, and what to keep in mind when reporting issues and opening pull requests.

3.1 Contribution guidelines

If you discovered bugs and issues, have ideas for improvements or new features, or want to contribute a new syntax checker, please report to the issue tracker the repository and send a pull request, but respect the following guidelines.

3.1.1 Reporting issues

- Check that the issue has not already been reported.
- Check that the issue has not already been fixed in the latest code.
- Be clear and precise (do not prose, but name functions and commands exactly).
- Include the version of sphinxcontrib-emacs.
- Open an issue with a clear title and description in grammatically correct, complete sentences.

3.1.2 Contributing code

Contributions of code, either as pull requests or as patches, are *very* welcome, but please respect the following guidelines.

General

- Write good and *complete* code.
- Provide use cases and rationale for new features.

Code style

- Respect **PEP 8**.
- Make sure that your code does not introduce Pylint warnings, using `.pylintrc` in the top-level source directory. Flycheck will help here.

- Add docstrings for every declaration, following **PEP 257**. Use Sphinx markup in docstrings.

Commit messages

Write commit messages according to Tim Pope's guidelines. In short:

- Start with a capitalized, short (50 characters or less) summary, followed by a blank line.
- If necessary, add one or more paragraphs with details, wrapped at 72 characters.
- Use present tense and write in the imperative: "Fix bug", not "fixed bug" or "fixes bug".
- Separate paragraphs by blank lines.
- Do *not* use special markup (e.g. reST or Markdown). Commit messages are plain text. You may use `*emphasis*` or `_underline_` though, following conventions established on mailing lists.

This is a model commit message:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Git Commit Mode and Magit provide a major mode for Git commit messages, which helps you to comply to these guidelines.

Pull requests

- Use a **topic branch** to easily amend a pull request later, if necessary.
- Do **not** open new pull requests, when asked to improve your patch. Instead, amend your commits with `git rebase -i`, and then update the pull request with `git push --force`
- Open a pull request that relates to but one subject with a clear title and description in grammatically correct, complete sentences.

CHAPTER 4

License

C

E

G

I

P

V