sphinx-litprog Documentation

Florian Brucker

Mar 25, 2019

Contents

A Sphinx extension for literate programming.

Literate programming is a programming technique which mixes formal source code and its explanation in natural language in the same document. *sphinx-litprog* allows you to embed source code directly into a Sphinx document and will automatically export that code into an external file from where it can be tested, used as a module, etc.

An example for this approach is *sphinx-litprog* itself, which is *developed using literate programming*.

CHAPTER 1

Installation

sphinx-litprog can be installed via `pip`:

```
pip install sphinx-litprog
```

After installing the extension, you need to activate it in your Sphinx project. To do this, add `'sphinx_litprog'` to the list of extensions in in your `conf.py`:

```
extensions = [
    ...
    'sphinx_litprog'
]
```

CHAPTER 2

Usage

Literate programming using *sphinx-litprog* consists of two steps. First, you use the `litprog` directive to embed source code in your Sphinx documents:

```
A simple implementation of the **Fibonacci sequence** in Python is
via recursion:

.. litprog::

    def fib(n):
        if n <= 2:
            return 1
        return fib(n - 1) + fib(n - 2)
```

You can use `litprog` directives in multiple documents, and multiple `litprog` directives per document.

When used with the default Sphinx builders (e.g. the HTML builder), the `litprog` directive produces the same output as the code-block directive.

Afterwards, use the `litprog` builder to extract the embedded source code into a separate file:

```
sphinx-build -b litprog /your/sphinx/project /output/directory
```

This will export the source code from the `litprog` directives into a file called `litprog.py` inside the output directory. You can change the name and the location of the exported file using the `litprog_filename` configuration setting in your `conf.py`:

```
# Path to the file containing the exported literate programming
# source code, relative to the output directory.
litprog_filename = some/other/path.py
```

Code from multiple `litprog` directives in the same document is exported in the order of the directives. Code from `litprog` directives in multiple documents is exported in the order of the documents according to the Table of Contents (as defined by Sphinx's `toctree` directive) in depth-first fashion.

The `litprog` directive supports the same arguments and options as the code-block directive. In addition, the `:hidden:` flag hides the content in the normal documentation output (but not in the exported source code).

CHAPTER 3

Change Log

Please refer to the file CHANGELOG.md.

CHAPTER 4

Development

sphinx-litprog itself is *developed using literal programming*. Version control and issue management happens on GitHub.

CHAPTER 5

The Extension

As an example, here is the source code of the *sphinx-litprog* extension itself in literate programming style.

The source code you see on this page is *not* copied from the `sphinx_litprog` Python module — the module is instead generated from the code on this page! That is, the *sphinx-litprog* extension is itself developed using literate programming.

For more details, take a look at the source code of this documentation page and at the automatically generated module code.

5.1 Module Header

Our module starts with its docstring, which documents its overall purpose:

```
'''
A literate programming extension for Sphinx.
'''
```

Next, in accordance with PEP8, we have the imports. We need

- `os.path` for generating the output filename,
- `docutils.parsers.rst.directives` for defining the options of our directive `LitProgDirective`,
- `sphinx.builders.Builder` as the superclass for `LitProgBuilder`, and
- `sphinx.directives.code.CodeBlock` as the superclass for `LitProgDirective`.

```
import os.path

from docutils.parsers.rst import directives
from sphinx.builders import Builder
from sphinx.directives.code import CodeBlock
```

We define the version of our module, using Semantic Versioning (see the change log for a history of changes):

```
__version__ = '0.1.1'
```

5.2 The `litprog` Directive

The first part of our extension is a custom reStructuredText directive which marks the source code portions of a literate programming document.

In the generated documentation, the content of the directive is displayed like in the `code-block` directive, therefore we extend the corresponding class `sphinx.directives.code.CodeBlock`.

```
class LitProgDirective(CodeBlock):
    '''
    Literate programming directive.

    Supports the same arguments/options as the ``code-block``
    directive.

    In addition, the ``:hidden:`` flag can be used to hide the
    content of the directive in the generated documentation (it will
    still be included in the exported literate programming source
    code).
    '''
    # In old Sphinx versions, the CodeBlock directive has a required
    # argument for specifying the programming language.
    required_arguments = 0

    option_spec = dict(CodeBlock.option_spec)
    option_spec['hidden'] = directives.flag

    def run(self):
        # Store content in environment for later export
        env = self.state.document.settings.env
        doc_snippets = _get_snippets(env).setdefault(env.docname, [])
        doc_snippets.extend(self.content)

        if 'hidden' in self.options:
            # Don't produce output in the documentation
            return []

        # Provide fake argument so that CodeBlock is happy in old
        # Sphinx versions
        self.arguments = ['python']

        # Delegate node creation to superclass
        return super().run()
```

The main part of that class is the `run`()` method, which is called when the directive is encountered while parsing a restructuredText document.

The job of `run`()` is to create the nodes which represent the directive's content in the document tree. We simply delegate that task to `CodeBlock.run()`, unless the `:hidden:` flag is set, in which case we return no nodes at all (so that the directive's content does not show up in the generated documentation).

Before doing that, however, we perform a crucial part of our extension's functionality: the raw content of the directive is stored in Sphinx's environment, from where it is later loaded by our builder when the literate programming source code is exported to a file.

We store the literate programming snippets from all restructuredText documents in a central datastructure that maps each document name to a list of lines. To initialize that data structure we use a little helper function:

```
def _get_snippets(env):
    '''
    Get the literate programming snippets from the environment.

    The snippets mapping is initialized if necessary.
    '''
    if not hasattr(env, 'litprog_snippets'):
        env.litprog_snippets = {}
    return env.litprog_snippets
```

5.3 The `litprog` Builder

The job of the builder is to take the source code snippets from Sphinx's environment and write them to a file in the correct order.

Like all Sphinx builders we inherit from `sphinx.builders.Builder`. Since our builder is not a typical builder like the ones for HTML or text output, most of our method implementations do nothing.

The method `Builder.get_outdated_docs()` is called by Sphinx to get a list of the documents whose output files for that builder are outdated. Since our builder does not have a 1-to-1 mapping between documents and output files we simply return a list of all documents.

```
class LitProgBuilder(Builder):
    name = 'litprog'

    def get_outdated_docs(self):
        return self.env.found_docs

    def get_target_uri(self, *args, **kwargs):
        return ''

    def prepare_writing(self, *args, **kwargs):
        return

    def write_doc(self, *args, **kwargs):
        return

    def finish(self):
        config = self.app.config
        env = self.app.env
        snippets = _get_snippets(env)
        filename = os.path.join(self.outdir, config.litprog_filename)
        with open(filename, 'w', encoding='utf-8') as f:
            for docname in _docnames_in_toc_order(env):
                doc_snippets = snippets.get(docname, [])
                if doc_snippets:
                    f.write('\n'.join(doc_snippets) + '\n')
```

The actual work is done in the `finish()` method: we iterate over all document names in depth-first order as defined by the `toctree` directive and write the corresponding source code snippets to a file.

The name of the output file is obtained from the `litprog_filename` configuration setting which we set up later on.

We use a generator function to provide the document names in the correct order:

```
def _docnames_in_toc_order(env):
    '''
    Yields all docnames in depth-first TOC order.
    '''
    includes = env.toctree_includes
    stack = [env.config.master_doc]
    while stack:
        docname = stack.pop()
        yield docname
        children = includes.get(docname, [])
        stack.extend(reversed(children))
```

5.4 Sphinx Integration

Now that we have implemented our directive and our builder we need to register them with Sphinx so that they can actually be used. This is done in the `setup` function, which Sphinx automatically calls for every extension listed in the `extensions` configuration setting.

```
def setup(app):
    app.add_builder(LitProgBuilder)
    app.add_directive('litprog', LitProgDirective)
    app.add_config_value('litprog_filename', 'litprog.py', '')
    app.connect('env-purge-doc', _purge_doc_snippets)
    return {
        'version': __version__,
        'env_version': 1,
        'parallel_read_safe': True,
        'parallel_write_safe': True,
    }
```

We register our builder, our directive, and the `litprog_filename` configuration option. In addition, we install a custom event handler for the `env-purge-doc` event. This allows us to clear the stored snippets for a given document when that document is removed or before it is parsed again:

```
def _purge_doc_snippets(app, env, docname):
    _get_snippets(env).pop(docname, None)
```