demo-Sphinx-autodoc

unknown

Sep 03, 2022

CONTENTS

This is a demo show-casing how to document a Python library with Sphinx, including the library's public API via the Autodoc and Autosummary extensions. It uses reStructuredText (reST) in its hand-written documents as well as in the doc-strings embedded with the library code.

The demo here is the yard stick to compare a Markdown-based workflow against:

- demo-MyST-docstring: Also uses the Sphinx documentation system, but has MyST parse the source files and doc-strings as Markdown instead of reST.
- demo-MkDocstrings: Uses MkDocs with the MkDocstrings plug-in. This is an entirely different documentation build system that uses Markdown throughout.

As for this demo, you can click on "Show Source" at the bottom of every page to see the reStructuredText from which it was rendered.

CHAPTER

ONE

OVERVIEW

Pretend this is the tutorial that gives a general introduction to the library, providing usage examples and all that.

This is a stand-alone document, in this case a file named `overview.rst` inside the project's `docs` folder. So it is separate from the actual Python library in the, unimaginatively named, `package` folder. Both folders are right underneath the project's root in the repo.

We have set up the *API* documentation as a different chapter. It is also a stand-alone document, named `api.rst`, and is linked in the side bar on the left. Readers can go there to understand how the library is to be used in application code. That is, it documents the *public* API. Not every doc-string defined in `package` needs to show up there, only the ones that are important. So we kick things off with a general summary of the top-level objects, courtesy of Sphinx's Autosummary extension, which links to in-depth API documentation provided by the Autodoc extension.

We can then link to objects from the API documentation, such as *`Class1`* or *`action`*. The syntax is just `` `Class1` `` and `` `action <package.action>` `` (as the latter reference happens to be ambiguous). This works as long as we have set `default_role = 'any'` in Sphinx's configuration file `conf.py`. We could also do that on a per-document basis with `.. default-role:: any`, but the `any` role is so useful, it rarely makes sense to assign anything else as the default.

Unless you want single back-ticks to denote `literals`, as they do in Markdown. Then you might configure `` `default_role = 'literal' ``, but would have to write `` :any:`Class1` `` to create a reference to the API documentation of *`Class1`*. So pick your poison.

Some people like to document the API within the general documentation as they go along. So instead of just referring to *`Class1`*, they pull in its doc-string somewhere in the text:

class Class1

This is the first line in the doc-string of `Class1`.

It is part of module *`classes`*.

action(*do='nothing'*)

This is a method of `Class1`.

Many projects also like to have Intersphinx references, so that they can easily link to, for example, Python's `pathlib` module. This needs to be set up in `conf.py`, but makes for shorter link targets.

One noteworthy shortcoming of using reST as a markup language is that we cannot have `literals`, or even *emphasis*, inside link text. That's because reStructuredText does not support nested markup of any kind. Note how this works (which is `` `this <https://example.org>`_ `` in the source), but `` `this <https://example.org>`_ `` is broken on this very page. Even though it's the same syntax as before, only with back-ticks around "this". (You'll have to click "Show Source" at the bottom of the page to see the original markup, as it isn't possible to reproduce it on the rendered page, at least not inline.)

1.1 First steps

This is a section inside the Overview chapter. We have marked it as a possible link target by putting `.. _first-steps:` right above the section header. We could also generate section labels automatically, once and for all, if we used the Autosectionlabel extension.

Here is a code example:

```
from package.classes import Class1

class1 = Class1()
class1.action()
```

This requires a `.. code-block::` directive, followed by a blank line, followed by the actual code indented one level. It's automatically highlighted in the colors defined by the theme, which in this case is Furo. Click the icon at the top right of the page to switch between dark and light mode and notice how the syntax highlighting changes along with it. We could easily replace Furo with any of a number of Sphinx themes. As themes are completely independent of the documentation semantics, all it takes is assigning another name to `html_theme` in `conf.py` (and `pip install`-ing the corresponding package).

CHAPTER

TWO

API

This is the front page for the API documentation. It uses the Sphinx extension Autosummary, which creates an overview page that links to individual pages created by the (separate) Autodoc extension.

action	This is the first line in the doc-string of function `action`.
actions	This is the first line in the doc-string of module `actions`.
classes	This is the first line in the doc-string of module `classes`.

2.1 action

action(*do='something'*)

This is the first line in the doc-string of function `action`.

It is defined in module *actions*.

2.2 actions

This is the first line in the doc-string of module `actions`.

We can reference other objects, such as *Class1* and *Class2*. We can link back to one of the main documents as a whole, for example *Overview*, or *a specific section*. We can create external cross-references like to `Path` thanks to the Intersphinx extension.

And we can have highlighted code examples:

```
from package import action
from package import Class1

action(do='whatever')
class1 = Class1()
class1.action()
```

Sphinx created this page from a "stub" file named `package.actions.rst` in the `api` folder underneath `docs`. As you can tell from clicking "Show Source" at the bottom of this very page, it contains very little:

```
actions
-------

.. automodule:: package.actions
```

Autodoc takes care of the rest and fills in the blanks, pulling in signatures and doc-strings from the package's source code. Autosummary would even create these stubs automatically, unless we tell it not to. We can also look at the source code of the `action` function, of this whole module in fact, if we click on the `[source]` link on the right, which is there courtesy of the Viewcode extension.

action(*do='something'*)

This is the first line in the doc-string of function `action`.

It is defined in module *actions*.

2.3 classes

This is the first line in the doc-string of module `classes`.

Here's a link to the more interesting module *actions*.

class Class1

This is the first line in the doc-string of `Class1`.

It is part of module *classes*.

action(*do='nothing'*)

This is a method of `Class1`.

class Class2

This is the first line in the doc-string of `Class2`.

It is also part of module *classes*.

action(*do=None*)

This is a method of `Class2`.

PYTHON MODULE INDEX

p

INDEX