sphinxcontrib-trio Documentation

Release 1.1.2

Nathaniel J. Smith

May 04, 2020

Contents

This sphinx extension helps you document Python code that uses async/await, or abstract methods, or context managers, or generators, or ... you get the idea. It works by making sphinx's regular directives for documenting Python functions and methods smarter and more powerful. The name is because it was originally written for the Trio project, and I'm not very creative. But don't be put off – there's nothing Trio- or async-specific about this extension; any Python project can benefit. (Though projects using async/await probably benefit the most, since sphinx's built-in tools are especially inadequate in this case.)

CHAPTER 1

Vital statistics

Requirements: This extension currently assumes you're using Python 3.5+ to build your docs. This could be relaxed if anyone wants to send a patch.

Documentation: https://sphinxcontrib-trio.readthedocs.io

Bug tracker and source code: https://github.com/python-trio/sphinxcontrib-trio

License: MIT or Apache 2, your choice.

Usage: `pip install -U sphinxcontrib-trio` in the same environment where you installed sphinx, and then add `"sphinxcontrib_trio"` to the list of `extensions` in your project's `conf.py`. (Notice that `"sphinxcontrib_trio"` has an underscore in it, NOT a dot. This is because I don't understand namespace packages, and I fear things that I don't understand.)

Code of conduct: Contributors are requested to follow our code of conduct in all project spaces.

CHAPTER 2

The big idea

Sphinx provides some convenient directives for documenting Python code: you can use the `method::` directive to document an ordinary method, the `classmethod::` directive to document a classmethod, the `decoratormethod::` directive to document a decorator method, and so on. But what if you have a classmethod that's also a decorator? And what if you want to document a project that uses some of Python's many interesting function types that Sphinx *doesn't* support, like async functions, abstract methods, generators, . . . ?

It would be possible to keep adding directive after directive for every possible type: `asyncmethod::`, `abstractmethod::`, `classmethoddecorator::`, `abstractasyncstaticmethod::` – you get the idea. But this quickly becomes silly. sphinxcontrib-trio takes a different approach: it enhances the basic `function::` and `method::` directives to accept options describing the attributes of each function/method, so you can write ReST code like:

```
.. method:: overachiever(arg1, ...)
   :abstractmethod:
   :async:
   :classmethod:

   This method is perhaps more complicated than it needs to be.
```

and you'll get rendered output like:

abstractmethod classmethod await overachiever(*arg1*, ...)
This method is perhaps more complicated than it needs to be.

While I was at it, I also enhanced the `sphinx.ext.autodoc` directives `autofunction::` and `automethod::` with new versions that know how to automatically detect many of these attributes, so you could just as easily have written the above as:

```
.. automethod:: overachiever
```

and it would automatically figure out that this was an abstract async classmethod by looking at your code.

And finally, I made the legacy `classmethod::` directive into an alias for:

```
.. method::
   :classmethod:
```

and similarly `staticmethod`, `decorator`, and `decoratormethod`, so dropping this extension into an existing sphinx project should be 100% backwards-compatible while giving sphinx new superpowers.

Basically, this is how sphinx ought to work in the first place. Perhaps in the future it will. But until then, this extension is pretty handy.

CHAPTER 3

The details

The following options are supported by the enhanced `function::` and `method::` directives, and some of them can be automatically detected if you use `autofunction::` / `automethod::`.

Option	Renders like	Autodetectable?
`:async:`	*await* **fn**()	yes!
`:decorator:`	@**fn**	no
`:with:`	*with* **fn**()	yes! (see below)
`:with: foo`	*with* **fn**() *as foo*	no
`:async-with:`	*async with* **fn**()	yes! (see below)
`:async-with: foo`	*async with* **fn**() *as foo*	no
`:for:`	*for* ... *in* **fn**()	yes! (see below)
`:for: foo`	*for foo in* **fn**()	no
`:async-for:`	*async for* ... *in* **fn**()	yes! (see below)
`:async-for: foo`	*async for foo in* **fn**()	no

There are also a few options that are specific to `method::`. They are:

Option	Renders like	Autodetectable?
`:abstractmethod:`	*abstractmethod* **fn**()	yes!
`:staticmethod:`	*staticmethod* **fn**()	yes!
`:classmethod:`	*classmethod* **fn**()	yes!

3.1 Autodetection heuristics

- `:with:` is autodetected for:
  - functions decorated with contextlib.contextmanager or contextlib2.contextmanager,
  - functions that have an attribute `__returns_contextmanager__` with a truthy value.
- `:async-with:` is autodetected for:

  - functions decorated with contextlib.asynccontextmanager,
  - functions that have an attribute `__returns_acontextmanager__` (note the `a`) with a truthy value.

- `:for:` is autodetected for generators.
- `:async-for:` is autodetected for async generators. The code supports both native async generators (in Python 3.6+) and those created by the async_generator library (in Python 3.5+).

As you can see, autodetection is necessarily a somewhat heuristic process. To reduce the rate of false positives, the autodetection code assumes that any given function will have at most one out of the following options: `:async:`, `:with:`, `:async-with:`, `:for:`, `:async-for:`. For example, this avoids the situation where a generator is decorated with `contextlib.contextmanager`, and sphinxcontrib-trio ends up applying both `:for:` and `:with:`.

But, despite our best attempts, it's possible that the heuristics will go wrong. Please do report any cases where this happens, but in the mean time you can work around the issue by using the `:no-auto-options:` option to disable option sniffing, and then add the correct options manually. For example, this code will pull out `some_function`'s signature and docstring from the source code, and then treat it as returning an async generator, regardless of its actual attributes.

```
.. autofunction:: some_function
   :no-auto-options:
   :async-for:
```

Another situation where this might be useful is if you have a function with a complicated calling convention that can't be summarized in one line. I can't really recommend writing such APIs, but if you need to document one, then `:no-auto-options:` can be used to tell sphinxcontrib-trio to stop being helpful, and then you can describe the full calling convention in the text.

CHAPTER 4

Examples

A regular async function:

```
.. function:: example_async_fn(...)
   :async:

   This is an example.
```

Renders as:

await example_async_fn(...)
    This is an example.

A context manager with a hint as to what's returned:

```
.. function:: open(file_name)
   :with: file_handle

   It's good practice to use :func:`open` as a context manager.
```

Renders as:

with open(*file_name*) **as file_handle**
    It's good practice to use *open()* as a context manager.

The auto versions of the directives also accept explicit options, which are appended to the automatically detected options. So if `some_method` is defined as a `abstractmethod` in the source, and you want to document that it should be used as a decorator, you can write:

```
.. automethod:: some_method
   :decorator:
```

and it will render like:

abstractmethod @some_method
    Here's some text automatically extracted from the method's docstring.

CHAPTER 5

Bugs and limitations

- Python supports defining abstract properties like:

```
@abstractmethod
@property
def some_property(...):
    ...
```

  But currently this extension doesn't help you document them. The difficulty is that for Sphinx, properties are "attributes", not "methods", and we don't currently hook the code for handling `attribute::` and `autoattribute::`. Maybe we should?

- When multiple options are combined, then we try to render them in a sensible way, but this does assume that you're giving us a sensible combination to start with. If you give sphinxcontrib-trio nonsense, then it will happily render nonsense. For example, this ReST:

```
.. function:: all_things_to_all_people(a, b)
   :with: x
   :async-with: y
   :for: z
   :decorator:

   Something has gone terribly wrong.
```

  renders as:

  with async with for z in @all_things_to_all_people(*a*, *b*) **as x as y**
  Something has gone terribly wrong.

- There's currently no particular support for asyncio's old-style "generator-based coroutines", though they might work if you remember to use asyncio.coroutine.

CHAPTER 6

Acknowledgements

Inspiration and hints on sphinx hackery were drawn from:

- sphinxcontrib-asyncio
- Curio's local customization
- CPython's local customization

sphinxcontrib-asyncio was especially helpful. Compared to sphinxcontrib-asyncio, this package takes the idea of directive options to its logical conclusion, steals Dave Beazley's idea of documenting special methods like coroutines by showing how they're used ("await f()" instead of "coroutine f()"), and avoids the forbidden word coroutine.

CHAPTER 7

Revision history

7.1 Sphinxcontrib_Trio 1.1.2 (2020-05-04)

7.1.1 Bugfixes

- Recent version of Sphinx deprecated its `PyClassmember` class. We've adjusted sphinxcontrib-trio's internals to stop using it and silence the warning. (#154)

7.2 Sphinxcontrib_Trio 1.1.1 (2020-03-26)

7.2.1 Bugfixes

- When using autodoc to document a class that has inherited members, we now correctly auto-detect the asyncness and other properties of those inherited methods. (#19)
- Recent versions of Sphinx deprecated its `PyModulelevel` class. We've adjusted sphinxcontrib-trio's internals to stop using it. (#138)

7.3 Sphinxcontrib_Trio 1.1.0 (2019-06-03)

7.3.1 Features

- Added support for Sphinx 2.1. (#23)

7.3.2 Deprecations and Removals

- Drop support for Sphinx 1.6 and earlier. (#87)

7.4 Sphinxcontrib_Trio 1.0.2 (2019-01-27)

7.4.1 Bugfixes

- Previously, on Sphinx 1.7, `autodoc_member_order="bysource"` didn't work correctly for async methods. Now, it does. (#13)

7.4.2 Deprecations and Removals

- Remove support for sphinx<1.6. (#14)

7.5 sphinxcontrib-trio 1.0.1 (2018-02-06)

7.5.1 Bugfixes

- Fix an obscure incompatibility with the `sphinx.ext.autosummary` module's `autosummary_generate = True` setting. (#8)
- Previously, sphinxcontrib-trio had to be listed after sphinx.ext.autodoc in your extensions configuration, or else sphinx would error out. Now sphinxcontrib-trio automatically loads sphinx.ext.autodoc as needed. (#9)

7.6 sphinxcontrib-trio v1.0.0 (2017-05-12)

Added autodetection heuristics for context managers.

Added rule to prevent functions using `@contextlib.contextmanager` or similar from being detected as generators (see bpo-30359).

Added `:no-sniff-options:` option for when the heuristics go wrong anyway.

Added a test suite, and fixed many bugs... but I repeat myself.

7.7 sphinxcontrib-trio v0.9.0 (2017-05-11)

Initial release.

Index