autoclasstoc

Release 1.3.0

unknown

Mar 02, 2022

GETTING STARTED

It's surprisingly difficult to document large Python classes in a way that's easy for users to navigate. Most projects use the `autodoc` Sphinx plugin, which simply puts the complete documentation for each class member one after another. While this does fully document the class, it doesn't give the user a quick way to see everything the class can do. This makes classes of even moderate complexity difficult to navigate. It also encourages projects to be stingy about which class members to include in the documentation (e.g. excluding special methods, inherited methods, private methods, and/or undocumented methods), to the further detriment of the user.

What's needed is for each class to have a succinct table of contents (TOC) that:

- Is organized into sections that will be meaningful to the user. Different projects and classes may call for different sections, e.g. public/private methods, methods that share a decorator, methods with a common prefix, etc.
- Includes every method of the class (so that the documentation is complete), while still making it easy for the user to get a sense for what the class does and find what they're looking for.
- Collapses inherited methods. Complex classes in particular can inherit a lot of methods from their parent classes, and while these methods should be present in the TOC (since they're part of the class), collapsing them makes it easier for the user to grok the functionality provided by the class itself.

`autoclasstoc` provides a new Restructured Text directive that is all of these things. It also works well with `autodoc` and `autogen`, and should be easy to incorporate into any existing project.

CHAPTER

ONE

ADVANCED USAGE

The following topics may be relevant when working on real-world documentation projects, which often demand a greater level of customization.

1.1 A TOC for every class

It is also possible to use *autoclasstoc* in auto-generated API documentation, i.e. where all of the classes in your project are documented without you having to explicitly write an `autoclass` directive for each one. The way to do this is to use the `sphinx.ext.autosummary` extension with a custom Jinja template for classes, as detailed below:

1. Configure the `sphinx.ext.autosummary` extension to automatically generate stub files each time the documentation is built:

   Listing 1: conf.py

   ```
   autosummary_generate = True
   ```

   Alternatively, you could generate stub files yourself by running the `autogen` command when necessary (after completing steps 2 and 3 below). I find this less convenient, but it might be better if you intend to edit the stub files by hand:

   ```
   $ sphinx-autogen -t _templates path/to/doc/with/autosummary.rst
   ```

2. Add an `autosummary` directive with the `:toctree:` and `:recursive:` options to your documentation. Anywhere will work, but `index.rst` is a common choice:

   Listing 2: index.rst

   ```
   .. autosummary::
      :toctree: path/to/directory/for/autogenerated/files
      :recursive:

      module.to.document
   ```

3. Provide a custom Jinja template for formatting class stub files. The purpose of this template is to specify that *autoclasstoc* should be used for each class:

   Listing 3: _templates/autosummary/class.rst

   ```
   {{ fullname | escape | underline}}
   ```

(continues on next page)

(continued from previous page)

```
.. currentmodule:: {{ module }}

.. autoclass:: {{ objname }}
   :members:
   :undoc-members:
   :special-members:
   :private-members:
   :inherited-members:
   :show-inheritance:

   .. autoclasstoc::
```

Note that the name of the `_templates` directory depends on the value of the `templates_path` setting in `conf.py`.

1.2 Custom sections

By default, *autoclasstoc* divides the TOC into sections based whether or not attributes are methods, and whether or not they are public. This is a reasonable default, but for many projects it may make sense to add custom sections specific to the idioms of that project. Fortunately, this is easy to configure. The basic steps are:

1. Define new *autoclasstoc.Section* subclasses.
2. Reference the subclasses either in `conf.py` or in the documentation itself.

This approach is very powerful, because the *Section* class controls all aspects of defining and formatting the TOC sections, and its subclasses can overwrite any of that behavior. Below are some specific examples showing how custom sections can be configured:

1.2.1 Based on name

Categorizing attributes based on their names is convenient, because it doesn't require making any changes or annotations to the code itself. For this example, we'll make a custom "Event Handlers" section that will consist of methods that begin with the prefix "on_", e.g. `on_mouse_down()` or `on_key_up()`.

The first step is to define a new *Section* subclass with the following attributes:

- *key*: used to include or exclude the section from class TOCs.
- *title*: how the section will be labeled in the documentation.
- *predicate()*: which attributes to include in the section.

Listing 4: conf.py

```
from autoclasstoc import Section, is_method

class EventHandlers(Section):
    key = 'event-handlers'
    title = "Event Handlers:"

    def predicate(self, name, attr, meta):
        return is_method(name, attr) and name.startswith('on_')
```

We also have to redefine the "Public Methods" section, so that it *doesn't* include the event handlers (as it otherwise would):

Listing 5: conf.py

```
from autoclasstoc import PublicMethods

class RemainingPublicMethods(PublicMethods):

    def predicate(self, name, attr, meta):
        return super().predicate(name, attr, meta) and not name.startswith('on_')
```

Finally, we need to specify that our new sections should be used by default (and what order they should go in):

Listing 6: conf.py

```
autoclasstoc_sections = [
        'event-handlers',
        'public-methods',
        'private-methods',
]
```

1.2.2 Based on decorator

A more explicit way to categorize methods is to use a decorator to label methods that belong to a particular section. This approach only is only applicable to methods and inner classes (because data attributes cannot be decorated), but is easy to implement. For this example, we'll make a section for "Read Only" methods that are identified by a decorator:

The first step is to write a decorator to label read-only methods:

```
def read_only(f):
    f.__readonly__ = True
    return f

class MyClass:

    @read_only
    def do_nothing(self):
        pass
```

Next, we have to define *Section* subclasses that are aware of the decorator:

Listing 7: conf.py

```
from autoclasstoc import Section

class ReadOnlySection(Section):
    key = 'read-only'
    title = "Read-Only Methods:"

    def predicate(self, name, attr, meta):
        return getattr(attr, '__readonly__', False)

class ReadWriteSection(Section):
```

(continues on next page)

(continued from previous page)

```
    key = 'read-write'
    title = "Read/Write Methods:"

    def predicate(self, name, attr, meta):
        return not getattr(attr, '__readonly__', False)

autoclasstoc_sections = [
        'read-only',
        'read-write',
]
```

Note that this example removes the distinction between private and public methods, so both the "Read-Only" and "Read/Write" sections will contain public and private methods.

1.2.3 Based on `:meta:` fields

With `sphinx.ext.autodoc`, it's possible to describe how an object should be documented by including `:meta:` fields in that object's docstring. *autoclasstoc* automatically parses these fields and provides them as an argument to *predicate()*, so they can be easily used to categorize attributes. As in the previous example, we'll make a custom section for read-only methods. The snippet below shows how such a method might be identified using a meta field:

```
class MyClass:

    def do_nothing(self):
        """
        This method doesn't do anything.

        :meta read-only:
        """
        pass
```

These meta fields are parsed into a dictionary such that `:meta key: value` would give `{'key': 'value'}`. This dictionary is provided to the *predicate()* method via the *meta* argument:

Listing 8: conf.py

```
from autoclasstoc import Section

class ReadOnlySection(Section):
    key = 'read-only'
    title = "Read-Only Methods:"

    def predicate(self, name, attr, meta):
        return 'read-only' in meta

class ReadWriteSection(Section):
    key = 'read-write'
    title = "Read/Write Methods:"

    def predicate(self, name, attr, meta):
        return 'read-only' not in meta
```

(continues on next page)

(continued from previous page)

```
autoclasstoc_sections = [
        'read-only',
        'read-write',
]
```

1.3 Custom CSS

All of the HTML elements generated by *autoclasstoc* are contained in a with class autoclasstoc. This can be used to select and style the elements in the class TOC. Note that the plugin includes some default rules to control the spacing around the <details> elements that contain TOCs for inherited attributes.

CHAPTER

TWO

GETTING HELP

If you find a bug, or need help getting `autoclasstoc` to work, please open a new issue on Github. Pull requests are also welcome!

CHAPTER

THREE

THIRD-PARTY PROJECTS

Below are links to third-party projects that use `autoclasstoc` in their documentation. Hopefully these examples are useful both to show what `autoclasstoc` looks like "in the wild", and to provide inspiration for your own documentation:

- Glooey
- KXG Game Engine

CHAPTER

FOUR

RESTRUCTURED TEXT

The following directive can be used in any restructured text file:

`.. autoclasstoc::` `[qualified class name]`
Create a table of contents (TOC) for the given Python class.

The TOC contains a link to each method defined in the given class. By default, the links are organized into four groups: "Public Attributes", "Public Methods", "Private Attributes", and "Private Methods". Public attributes/methods are those with names that either don't begin with an underscore or begin and end with two underscores (i.e. "dunder methods"). Every other attribute/method is private. It's easy to define custom sections; see *Advanced Usage* for more details.

In addition to attributes directly defined in the given class, the TOC will also include links to inherited attributes. These links are grouped by the class they are inherited from, and are collapsed by default to keep the TOC succinct.

The `[qualified class name]` argument is optional if this directive occurs within an `autoclass` or a `py:class` directive (in which case the class name can be inferred from the context). If this argument is provided, it must be the full name of the class, in the same manner expected by `autoclass`.

`:sections:`
A comma-separated list of sections to include in the class TOC. If specified, this supercedes the *autoclasstoc_sections* setting from `conf.py`.

`:exclude-sections:`
A comma-separated list of sections to exclude from the class TOC. This can be used in conjunction with the `:sections:` option above. No sections are excluded by default.

Note: The "class TOCs" created by this directive are not related to the TOCs defined by `toctree`. The term TOC is just used to mean an organized list of links to more detailed documentation.

CHAPTER

FIVE

SPHINX CONFIGURATION

The following setting can be defined in `conf.py`:

autoclasstoc_sections

The default list of sections to include in class TOCs, in the order they should appear in the documentation. The values in the list can be either strings or *Section* classes. Strings are the same values that can be provided to the section options of the *autoclasstoc* directive, and must refer to the name of a section class. The following section names are available by default:

public-methods Methods that don't begin with an underscore (or that are special methods, e.g. `__init__()`).

private-methods Methods that do begin with an underscore (and are not special).

public-attrs Non-methods and non-classes that don't begin with an underscore.

private-attrs Non-methods and non-classes that begin with an underscore.

inner-classes Classes defined within the class in question.

The names of any *custom sections* that have been defined can be used as well. The default value for this setting is:

```
autoclasstoc_sections = [
        'public-attrs',
        'public-methods',
        'private-attrs',
        'private-methods',
]
```

CHAPTER

SIX

PYTHON API

The primary purpose of the Python API is to define *custom sections* for the class TOCs. This is done by subclassing the `Section` class, although a number of helper functions and classes are also available.

autoclasstoc.AutoClassToc(name, arguments, ...)	Generate a succinct TOC for automatically documented classes.
autoclasstoc.Section(state, cls)	Format a specific section in a class TOC, e.g.
autoclasstoc.PublicMethods(state, cls)	Include a "Public Methods" section in the class TOC.
autoclasstoc.PrivateMethods(state, cls)	Include a "Private Methods" section in the class TOC.
autoclasstoc.PublicDataAttrs(state, cls)	Include a "Public Data Attributes" section in the class TOC.
autoclasstoc.PrivateDataAttrs(state, cls)	Include a "Private Data Attributes" section in the class TOC.
autoclasstoc.InnerClasses(state, cls)	Include an "Inner Classes" section in the class TOC.
autoclasstoc.is_method(name, attr)	Return true if the given attribute is a method or property.
autoclasstoc.is_data_attr(name, attr[, ...])	Return true if the given attribute is a data attribute, e.g.
autoclasstoc.is_public(name)	Return true if the given name is public.
autoclasstoc.is_private(name)	Return true if the given name is private.
autoclasstoc.is_special(name)	Return True if the name starts and ends with a double-underscore.
autoclasstoc.utils	
autoclasstoc.nodes	The `autoclasstoc` module defines two new *docutils* nodes, which make it possible to create collapsible content in HTML.
autoclasstoc.ConfigError	Indicate an configuration error affecting `autoclasstoc`.

6.1 autoclasstoc.Section

class autoclasstoc.**Section**(*state*, *cls*)

Format a specific section in a class TOC, e.g. “Public Methods”.

The purpose of this class is to make it easy to customize the sections that make up the class TOC. For example, you might want an “Event Handler” section that includes any method that starts with “on_”. Or you might want to format the links in a table with multiple columns, to save more space.

These kinds of things can be accomplished by subclassing `Section` and overwriting the relevant methods. Almost every method is meant to be overridden by subclasses, but most subclasses will only need to override `key`, `title`, and `predicate`. `key` and `title` have no default value, and must be overridden in each subclass.

predicate determines which attributes are included in the section, which is the primary purpose of most custom sections.

Public Data Attributes:

key
title
include_inherited

Public Methods:

__init__(state, cls)	Create a section for a specific class.
__init_subclass__()	Keep track of any *Section* subclasses that are defined.
check()	Raise *ConfigError* if the section has not been configured correctly, e.g.
format()	Return a list of *docutils* nodes that will compose the section.
predicate(name, attr, meta)	Return true if the given attribute should be included in this section.

Private Methods:

_make_container()	Create the container node that will contain the entire section.
_make_rubric()	Create the section header node.
_make_links(attrs)	Make a link to the full documentation for each attribute.
_make_inherited_details(parent)	Make a collapsible node to contain links to inherited attributes.
_filter_attrs(attrs)	Return only those attributes that match the predicate.
_find_attrs()	Return all attributes associated with this class.
_find_inherited_attrs()	Find attributes that this class has inherited from other classes.

Full Documentation:

__annotations__ = {}

__init__(*state*, *cls*)
Create a section for a specific class.

Parameters

- **state** (*docutils.parsers.rst.states.RSTState*) – The state object associated with the *autoclasstoc* directive. This can be used to evaluate restructured text markup using *nodes_from_rst()*.
- **cls** (*type*) – The class to make the TOC section for.

classmethod __init_subclass__()
Keep track of any *Section* subclasses that are defined.

_filter_attrs(*attrs*)
Return only those attributes that match the predicate.

Parameters **attrs** (*dict*) – A dictionary of attributes, in the same format as __dict__.

Returns A dictionary in the same format as *attrs*.

_find_attrs()
Return all attributes associated with this class.

These attributes will subsequently be filtered to remove any that aren't relevant to this section, so there is no need to do any filtering here. The return value should be a name-to-attribute dictionary in the same format as __dict__.

_find_inherited_attrs()
Find attributes that this class has inherited from other classes.

These attributes will subsequently be filtered to remove any that aren't relevant to this section, so there is no need to do any filtering here. The return value should be a dictionary mapping parent class types to __dict__ style dictionaries.

_make_container()
Create the container node that will contain the entire section.

This method is meant to be overridden in subclasses. The primary purpose of the container node is to belong to a CSS class that can then be used to identify HTML elements associated with *autoclasstoc*.

_make_inherited_details(*parent*)
Make a collapsible node to contain links to inherited attributes.

This method is meant to be overridden in subclasses. The default implementation returns a `details` node, which is rendered in HTML as a <details> element.

_make_links(*attrs*)
Make a link to the full documentation for each attribute.

This method is meant to be overridden in subclasses. The default implementation creates the links using an `autosummary` directive.

Parameters **attrs** (*dict*) – A dictionary of attributes, in the same format as __dict__.

_make_rubric()
Create the section header node.

This method is meant to be overridden in subclasses.

check()
Raise *ConfigError* if the section has not been configured correctly, e.g. if it doesn't have a title specified.

format()
Return a list of *docutils* nodes that will compose the section.

The default implementation of this method creates and populates `autosummary` directives for the class in question and all of its superclasses. Almost all of

include_inherited = True

key = None

predicate(*name*, *attr*, *meta*)
Return true if the given attribute should be included in this section.

Parameters

- **name** (*str*) – The name of the attribute. In most cases, this is identical to `attr.__name__`.
- **attr** (*object*) – The attribute object itself.
- **meta** (*dict*) – Any :meta: fields present in the attribute's docstring, as parsed by `sphinx.util.docstrings.extract_metadata()`.

See also:

is_method *is_data_attr* *is_public* *is_private* *is_special*

title = None

6.2 autoclasstoc.is_method

autoclasstoc.**is_method**(*name*, *attr*)
Return true if the given attribute is a method or property.

6.3 autoclasstoc.is_data_attr

autoclasstoc.**is_data_attr**(*name*, *attr*, *exclude_special=True*)
Return true if the given attribute is a data attribute, e.g. not a method or an inner class. Many data attributes are properties.

By default, attributes with double-underscore names (e.g. `__dict__`) are not considered data attributes. Unlike special methods, these "special attributes" are very rarely relevant to users of a class. This behavior can be disabled by toggling the *exclude_special* argument.

6.4 autoclasstoc.is_public

autoclasstoc.**is_public**(*name*)
Return true if the given name is public.

Specifically, a name is public if it either doesn't start with an underscore, or if it starts and ends with two underscores (i.e. a "special" method).

6.5 autoclasstoc.is_private

autoclasstoc.**is_private**(*name*)

Return true if the given name is private.

A name is private if it starts with an underscore, but does not start and end with two underscores (i.e. not a special method).

6.6 autoclasstoc.is_special

autoclasstoc.**is_special**(*name*)

Return True if the name starts and ends with a double-underscore.

Such names typically have special meaning to Python, e.g. `__init__()`.

6.7 autoclasstoc.utils

Functions

comma_separated_list(x)	Parse a restructured text option as a comma-separated list of strings.
filter_attrs(attrs, predicate)	Remove attributes for which the given predicate function returns False.
find_inherited_attrs(cls)	Return a dictionary mapping parent classes to the attributes inherited from those classes.
load_class(mod_name, cls_name)	Import the given class from the given module.
make_container()	Make a container node to identify elements associated with the *autoclasstoc* directive.
make_inherited_details(state, parent[, ...])	Make a collapsible node to contain information about inherited attributes.
make_links(state, attrs)	Make links to the given class attributes.
make_rubric(title)	Make an informal header.
make_toc(state, cls, sections)	Create the class TOC.
nodes_from_rst(state, rst)	Create nodes from the given restructured text.
pick_class(qual_name, env)	Figure out which class to make the TOC for.
pick_sections(sections[, exclude])	Determine which sections to include in the class TOC.
strip_p(nodes)	Remove any top-level paragraph nodes.

6.7.1 autoclasstoc.utils.comma_separated_list

autoclasstoc.utils.**comma_separated_list**(*x*)
Parse a restructured text option as a comma-separated list of strings.

6.7.2 autoclasstoc.utils.filter_attrs

autoclasstoc.utils.**filter_attrs**(*attrs*, *predicate*)
Remove attributes for which the given predicate function returns False.

6.7.3 autoclasstoc.utils.find_inherited_attrs

autoclasstoc.utils.**find_inherited_attrs**(*cls*)
Return a dictionary mapping parent classes to the attributes inherited from those classes.

6.7.4 autoclasstoc.utils.load_class

autoclasstoc.utils.**load_class**(*mod_name*, *cls_name*)
Import the given class from the given module.

6.7.5 autoclasstoc.utils.make_container

autoclasstoc.utils.**make_container**()
Make a container node to identify elements associated with the *autoclasstoc* directive.

6.7.6 autoclasstoc.utils.make_inherited_details

autoclasstoc.utils.**make_inherited_details**(*state*, *parent*, *open_by_default=False*)
Make a collapsible node to contain information about inherited attributes.

6.7.7 autoclasstoc.utils.make_links

autoclasstoc.utils.**make_links**(*state*, *attrs*)
Make links to the given class attributes.

More specifically, the links are made using the `autosummary` directive.

6.7.8 autoclasstoc.utils.make_rubric

autoclasstoc.utils.**make_rubric**(*title*)
Make an informal header.

6.7.9 autoclasstoc.utils.make_toc

autoclasstoc.utils.**make_toc**(*state*, *cls*, *sections*)
Create the class TOC.

6.7.10 autoclasstoc.utils.nodes_from_rst

autoclasstoc.utils.**nodes_from_rst**(*state*, *rst*)
Create nodes from the given restructured text.

The *rst* argument can either be any of the following types:

- string, with any number of lines
- list of strings
- StringList (the type used by docutils to represent lines of restructured text)
- node

6.7.11 autoclasstoc.utils.pick_class

autoclasstoc.utils.**pick_class**(*qual_name*, *env*)
Figure out which class to make the TOC for.

We can either be given this information as an argument, or we can try to figure it out from the context (e.g. the `autoclass` or `py:class` currently being processed).

Parameters

- **qual_name** (*str*) – The name of the class to pick, or None if the class should be inferred from the environment.
- **env** (*sphinx.environment.BuildEnvironment*) – This object is available as `self.env` from `SphinxDirective` subclasses.

6.7.12 autoclasstoc.utils.pick_sections

autoclasstoc.utils.**pick_sections**(*sections*, *exclude=None*)
Determine which sections to include in the class TOC.

The return value will be a list in the same order as *sections*, but with any sections from *exclude* removed. Both arguments can specify sections using string names (e.g. "public-methods") or un-instantiated *Section* classes. All names will be converted to classes in the return value.

6.7.13 autoclasstoc.utils.strip_p

autoclasstoc.utils.**strip_p**(*nodes*)
Remove any top-level paragraph nodes.

Parsing a simple string like "Hello world" with *nodes_from_rst* will return text wrapped in a paragraph. If this paragraph is not desired (e.g. because it messes with formatting), this function can be used to get rid of it.

6.8 autoclasstoc.nodes

The *autoclasstoc* module defines two new *docutils* nodes, which make it possible to create collapsible content in HTML.

Classes

`details`([open_by_default])	A node that can be expanded or collapsed by the user.
`details_summary`([rawsource, text])	The summary text to display when a `details` node is collapsed.

Functions

setup(app)	Configure Sphinx to use the `details` and `details_summary` nodes.

6.8.1 autoclasstoc.nodes.setup

autoclasstoc.nodes.**setup**(*app*)
Configure Sphinx to use the `details` and `details_summary` nodes.

6.9 autoclasstoc.ConfigError

exception autoclasstoc.**ConfigError**
Indicate an configuration error affecting *autoclasstoc*.

PYTHON MODULE INDEX

a

INDEX

Symbols

A

C

F

I

K

L

M

N

P

S

T