pdoc Documentation

Contents

Module pdoc

Python package pdoc provides types, functions, and a command-line interface for accessing public documentation of Python modules, and for presenting it in a user-friendly, industry-standard open format. It is best suited for small- to medium-sized projects with tidy, hierarchical APIs.

pdoc extracts documentation of:

- modules (including submodules),
- functions (including methods, properties, coroutines ...),
- classes, and
- variables (including globals, class variables, and instance variables).

Documentation is extracted from live objects' docstrings[1] using Python's `__doc__` attribute[2]. Documentation for variables is found by examining objects' abstract syntax trees.

What objects are documented?

pdoc only extracts *public API* documentation.[3] All objects (modules, functions, classes, variables) are only considered public if their *identifiers don't begin with an underscore* (_).[4]

In addition, if a module defines `__all__`[6], then only the identifiers contained in this list will be considered public. Otherwise, a module's global identifiers are considered public only if they don't begin with an underscore and are defined in this exact module (i.e. not imported from somewhere else).

By transitivity, sub-objects of non-public objects (e.g. submodules of non-public modules, methods of non-public classes etc.) are not public and thus not documented.

Where does pdoc get documentation from?

In Python, objects like modules, functions, classes, and methods have a special attribute `__doc__` which contains that object's documentation string (docstring[7]). For example, the following code defines a function with a docstring and shows how to access its contents:

```
>>> def test():
...     """This is a docstring."""
...     pass
...
>>> test.__doc__
'This is a docstring.'
```

It's pretty much the same with classes and modules. See PEP-257[8] for Python docstring conventions.

These docstrings are set as descriptions for each module, class, function, and method listed in the documentation produced by pdoc.

pdoc extends the standard use of docstrings in Python in two important ways: by allowing methods to inherit docstrings, and by introducing syntax for docstrings for variables.

[1] https://docs.python.org/3/glossary.html#term-docstring

[2] Documented modules are executed in order to provide `__doc__` attributes. Any non-fenced global code in imported modules will affect the current environment.

[3] Here, public API refers to the API that is made available to your project end-users, not the public API e.g. of a private class that can be reasonably extended elsewhere by your project developers.

[4] Prefixing private, implementation-specific objects with an underscore is a common convention[5].

[6] https://docs.python.org/3/tutorial/modules.html#importing-from-a-package

[7] https://docs.python.org/3/glossary.html#term-docstring

[8] https://www.python.org/dev/peps/pep-0257/

Docstrings inheritance

pdoc considers methods' docstrings inherited from superclass methods', following the normal class inheritance patterns. Consider the following code example:

```
>>> class A:
...     def test(self):
...         """Docstring for A."""
...         pass
...
>>> class B(A):
...     def test(self):
...         pass
...
>>> A.test.__doc__
'Docstring for A.'
>>> B.test.__doc__
None
```

In Python, the docstring for `B.test` doesn't exist, even though a docstring was defined for `A.test`. When pdoc generates documentation for the code such as above, it will automatically attach the docstring for `A.test` to `B.test` if `B.test` does not define its own docstring. In the default HTML template, such inherited docstrings are greyed out.

Docstrings for variables

Python by itself doesn't allow docstrings attached to variables[9]. However, pdoc supports docstrings attached to module (or global) variables, class variables, and object instance variables; all in the same way as proposed in PEP-224[10], with a docstring following the variable assignment. For example:

```
module_variable = 1
"""Docstring for module_variable."""

class C:
    class_variable = 2
    """Docstring for class_variable."""

    def __init__(self):
        self.variable = 3
        """Docstring for instance variable."""
```

While the resulting variables have no `__doc__` attribute, pdoc compensates by reading the source code (when available) and parsing the syntax tree.

By convention, variables defined in a class' `__init__` method and attached to `self` are considered and documented as instance variables.

Class and instance variables can also inherit docstrings.

Overriding docstrings with `__pdoc__`

Docstrings for objects can be disabled or overridden with a special module-level dictionary `__pdoc__`. The *keys* should be string identifiers within the scope of the module or, alternatively, fully-qualified reference names. E.g. for instance variable `self.variable` of class `C`, its module-level identifier is `'C.variable'`.

If `__pdoc__[key] = False`, then `key` (and its members) will be **excluded from the documentation** of the module.

[9] http://www.python.org/dev/peps/pep-0224
[10] http://www.python.org/dev/peps/pep-0224

Alternatively, the *values* of `__pdoc__` should be the overriding docstrings. This particular feature is useful when there's no feasible way of attaching a docstring to something. A good example of this is a namedtuple[11]:

```
__pdoc__ = {}

Table = namedtuple('Table', ['types', 'names', 'rows'])
__pdoc__['Table.types'] = 'Types for each column in the table.'
__pdoc__['Table.names'] = 'The names of each column in the table.'
__pdoc__['Table.rows'] = 'Lists corresponding to each row in the table.'
```

pdoc will then show `Table` as a class with documentation for the `types`, `names` and `rows` members.

Note: The assignments to `__pdoc__` need to be placed where they'll be executed when the module is imported. For example, at the top level of a module or in the definition of a class.

Supported docstring formats

Currently, pure Markdown (with extensions[12]), numpydoc[13], and Google-style[14] docstrings formats are supported. Basic reST directives (such as e.g. `.. versionadded::`, `.. deprecated::`, `.. note::`, `.. image::`, ...) should also work.

Linking to other identifiers

In your documentation, you may refer to other identifiers in your modules. When exporting to HTML, linking is automatically done whenever you surround an identifier with backticks[15] (` ). The identifier name must be fully qualified, for example `pdoc.Doc.docstring` is correct (and will link to `Doc.docstring`) while `Doc.docstring` is *not*.

Command-line interface

pdoc includes a feature-rich "binary" program for producing HTML and plain text documentation of your modules. To produce HTML documentation of your whole package in subdirectory 'build' of the current directory, using the default HTML template, run:

```
$ pdoc --html --html-dir build my_package
```

To run a local HTTP server while developing your package or writing docstrings for it, run:

```
$ pdoc --http : my_package
```

To re-build documentation as part of your continuous integration (CI) best practice, i.e. ensuring all reference links are correct and up-to-date, make warnings error loudly by settings the environment variable `PYTHONWARNINGS`[16] before running pdoc:

```
$ export PYTHONWARNINGS='error::UserWarning'
```

For brief usage instructions, type:

```
$ pdoc --help
```

Programmatic usage

The main entry point is `Module` which wraps a module object and recursively imports and wraps any submodules and their members.

[11] https://docs.python.org/3/library/collections.html#collections.namedtuple
[12] https://python-markdown.github.io/extensions/#officially-supported-extensions
[13] https://numpydoc.readthedocs.io/
[14] http://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings
[15] https://en.wikipedia.org/wiki/Grave_accent#Use_in_programming
[16] https://docs.python.org/3/using/cmdline.html#envvar-PYTHONWARNINGS

After all related modules are wrapped (related modules are those that share the same `Context`), you need to call `link_inheritance()` with the used `Context` instance to establish class inheritance links.

Afterwards, you can use `Module.html()` and `Module.text()` methods to output documentation in the desired format. For example:

```
import pdoc

modules = ['a', 'b']  # Public submodules are auto-imported
context = pdoc.Context()

modules = [pdoc.Module(mod, context=context)
           for mod in modules]
pdoc.link_inheritance(context)

def recursive_htmls(mod):
    yield mod.name, mod.html()
    for submod in mod.submodules():
        yield from recursive_htmls(submod)

for mod in modules:
    for module_name, html in recursive_htmls(mod):
        ...  # Process
```

When documenting a single module, you might find functions `html()` and `text()` handy. For importing arbitrary modules/files, use `import_module()`.

Alternatively, use the runnable script included with this package.

Custom templates

To override the built-in HTML/CSS and plain text templates, copy the relevant templates from `pdoc/templates` directory into a directory of your choosing and edit them. When you run pdoc command afterwards, pass the directory path as a parameter to the `--template-dir` switch.

Tip: If you find you only need to apply minor alterations to the HTML template, see if you can do so by overriding just some of the following, placeholder sub-templates:

- *config.mako*: Basic template configuration, affects the way templates are rendered.
- *head.mako*: Included just before `</head>`. Best for adding resources and styles.
- *logo.mako*: Included at the very top of the navigation sidebar. Empty by default.
- *credits.mako*: Included in the footer, right before pdoc version string.

See default template files for reference.

If working with `pdoc` programmatically, *prepend* the directory with modified templates into the `directories` list of the `tpl_lookup` object.

Compatibility

`pdoc` requires Python 3.5+. The last version to support Python 2.x is pdoc3 0.3.x[17].

Contributing

`pdoc` is on GitHub[18]. Bug reports and pull requests are welcome.

[17] https://pypi.org/project/pdoc3/0.3.11/
[18] https://github.com/pdoc3/pdoc

License

pdoc is licensed under the terms of GNU AGPL-3.0[19] {: rel=license} or later, meaning you can use it for any reasonable purpose and remain in complete ownership of all the documentation you produce, but you are also encouraged to make sure any upgrades to pdoc itself find their way back to the community.

Variables

Variable `tpl_lookup`

A `mako.lookup.TemplateLookup` object that knows how to load templates from the file system. You may add additional paths by modifying the object's `directories` attribute.

Functions

Function `html`

```
def html(module_name, docfilter=None, external_links=False, link_prefix='',
source=True, **kwargs)
```

Returns the documentation for the module `module_name` in HTML format. The module must be a module or an importable string.

`docfilter` is an optional predicate that controls which documentation objects are shown in the output. It is a function that takes a single argument (a documentation object) and returns `True` or `False`. If `False`, that object will not be documented.

If `external_links` is `True`, then identifiers to external modules are always turned into links.

If `link_prefix` is a non-empty string, all links will be relative to the top module and will have that prefix. Otherwise, all links will be relative.

If `source` is `True`, then source code will be retrieved, and outputted, for every Python object whenever possible. This can dramatically decrease performance when documenting large modules.

Function `import_module`

```
def import_module(module)
```

Return module object matching `module` specification (either a python module path or a filesystem path to file/directory).

Function `link_inheritance`

```
def link_inheritance(context=None)
```

Link inheritance relationsships between `Class` objects (and between their members) of all `Module` objects that share the provided `context` (`Context`).

You need to call this if you expect `Doc.inherits` and inherited `Doc.docstring` to be set correctly.

Function `reset`

```
def reset()
```

Resets the global `Context` to the initial (empty) state.

[19] https://www.gnu.org/licenses/agpl-3.0.html

Function `text`

```
def text(module_name, docfilter=None, **kwargs)
```

Returns the documentation for the module `module_name` in plain text format suitable for viewing on a terminal. The module must be a module or an importable string.

`docfilter` is an optional predicate that controls which documentation objects are shown in the output. It is a function that takes a single argument (a documentation object) and returns `True` or `False`. If `False`, that object will not be documented.

Classes

Class `Class`

Representation of a class' documentation.

Ancestors (in MRO)

- pdoc.__init__.Doc

Instance variables

Variable `doc`

A mapping from identifier name to a `Doc` objects.

Variable `refname`

Reference name of this documentation object, usually its fully qualified path (e.g. pdoc.Doc.refname). Every documentation object provides this property.

Methods

Method `__init__`

```
def __init__(self, name, module, obj, *, docstring=None)
```

Initializes a documentation object, where `name` is the public identifier name, `module` is a `Module` object where raw Python object `obj` is defined, and `docstring` is its documentation string. If `docstring` is left empty, it will be read with `inspect.getdoc()`.

Method `class_variables`

```
def class_variables(self, include_inherited=True)
```

Returns a sorted list of `Variable` objects that represent this class' class variables.

Method `functions`

```
def functions(self, include_inherited=True)
```

Returns a sorted list of `Function` objects that represent this class' static functions.

Method `inherited_members`

```
def inherited_members(self)
```

Returns all inherited members as a list of tuples (ancestor class, list of ancestor class' members sorted by name), sorted by MRO.

Method `instance_variables`

```
def instance_variables(self, include_inherited=True)
```

Returns a sorted list of `Variable` objects that represent this class' instance variables. Instance variables are those defined in a class's `__init__` as `self.variable = ...`.

Method `methods`

```
def methods(self, include_inherited=True)
```

Returns a sorted list of `Function` objects that represent this class' methods.

Method `mro`

```
def mro(self, only_documented=False)
```

Returns a list of ancestor (superclass) documentation objects in method resolution order.

The list will contain objects of type `Class` if the types are documented, and `External` otherwise.

Method `subclasses`

```
def subclasses(self)
```

Returns a list of subclasses of this class that are visible to the Python interpreter (obtained from `type.__subclasses__()`).

The objects in the list are of type `Class` if available, and `External` otherwise.

Class `Context`

The context object that maps all documented identifiers (`Doc.refname`) to their respective `Doc` objects.

You can pass an instance of `Context` to `Module` constructor. All `Module` objects that share the same `Context` will see (and be able to link in HTML to) each other's identifiers.

If you don't pass your own `Context` instance to `Module` constructor, a global context object will be used.

Ancestors (in MRO)

- builtins.dict

Class `Doc`

A base class for all documentation objects.

A documentation object corresponds to *something* in a Python module that has a docstring associated with it. Typically, this includes modules, classes, functions, and methods. However, `pdoc` adds support for extracting some docstrings from abstract syntax trees, making (module, class or instance) variables supported too.

A special type of documentation object `External` is used to represent identifiers that are not part of the public interface of a module. (The name "External" is a bit of a misnomer, since it can also correspond to unexported members of the module, particularly in a class's ancestor list.)

Descendants

- pdoc.__init__.Module
- pdoc.__init__.Class
- pdoc.__init__.Variable
- pdoc.__init__.Function
- pdoc.__init__.External

Instance variables

Variable `docstring`

The cleaned docstring for this object.

Variable `inherits`

The Doc object (Class, Function, or Variable) this object inherits from, if any.

Variable `module`

The module documentation object that this object is defined in.

Variable `name`

The identifier name for this object.

Variable `obj`

The raw python object.

Variable `qualname`

Module-relative "qualified" name of this documentation object, used for show (e.g. Doc.qualname).

Variable `refname`

Reference name of this documentation object, usually its fully qualified path (e.g. pdoc.Doc.refname). Every documentation object provides this property.

Variable `source`

Cleaned (dedented) source code of the Python object. If not available, an empty string.

Methods

Method `__init__`

```
def __init__(self, name, module, obj, docstring=None)
```

Initializes a documentation object, where `name` is the public identifier name, `module` is a `Module` object where raw Python object `obj` is defined, and `docstring` is its documentation string. If `docstring` is left empty, it will be read with `inspect.getdoc()`.

Method `url`

```
def url(self, relative_to=None, *, link_prefix='', top_ancestor=False)
```

Canonical relative URL (including page fragment) for this documentation object.

Specify `relative_to` (a `Module` object) to obtain a relative URL.

For usage of `link_prefix` see `html()`.

If `top_ancestor` is `True`, the returned URL instead points to the top ancestor in the object's `Doc.inherits` chain.

Class `External`

A representation of an external identifier. The textual representation is the same as an internal identifier.

External identifiers are also used to represent something that is not documented but appears somewhere in the public interface (like the ancestor list of a class).

Ancestors (in MRO)

- pdoc.__init__.Doc

Methods

Method `__init__`

```
def __init__(self, name)
```

Initializes an external identifier with `name`, where `name` should be a fully qualified name.

Method `url`

```
def url(self, *args, **kwargs)
```

`External` objects return absolute urls matching `/{name}.ext`.

Class `Function`

Representation of documentation for a function or method.

Ancestors (in MRO)

- pdoc.__init__.Doc

Instance variables

Variable `cls`

The `Class` documentation object if the function is a method. If not, this is None.

Variable `method`

Whether this function is a normal bound method.

In particular, static and class methods have this set to False.

Variable `refname`

Reference name of this documentation object, usually its fully qualified path (e.g. pdoc.Doc.refname). Every documentation object provides this property.

Methods

Method `__init__`

```
def __init__(self, name, module, obj, *, cls=None, method=False)
```

Same as `Doc.__init__()`, except `obj` must be a Python function object. The docstring is gathered automatically.

`cls` should be set when this is a method or a static function beloing to a class. `cls` should be a `Class` object.

`method` should be `True` when the function is a method. In all other cases, it should be `False`.

Method `funcdef`

```
def funcdef(self)
```

Generates the string of keywords used to define the function, for example `def` or `async def`.

Method `params`

```
def params(self)
```

Returns a list where each element is a nicely formatted parameter of this function. This includes argument lists, keyword arguments and default values, and it doesn't include any optional arguments whose names begin with an underscore.

Class `Module`

Representation of a module's documentation.

Ancestors (in MRO)

- pdoc.__init__.Doc

Instance variables

Variable `doc`

A mapping from identifier name to a documentation object.

Variable `is_package`

`True` if this module is a package.

Works by checking whether the module has a `__path__` attribute.

Variable `supermodule`

The parent `Module` this module is a submodule of, or `None`.

Methods

Method `__init__`

```
def __init__(self, module, *, docfilter=None, supermodule=None, context=None)
```

Creates a `Module` documentation object given the actual module Python object.

`docfilter` is an optional predicate that controls which sub-objects are documentated (see also: `html()`).

`supermodule` is the parent `Module` this module is a submodule of.

`context` is an instance of `Context`. If `None` a global context object will be used.

Method `classes`

```
def classes(self)
```

Returns all documented module-level classes in the module sorted alphabetically as a list of `Class`.

Method `find_class`

```
def find_class(self, cls)
```

Given a Python `cls` object, try to find it in this module or in any of the exported identifiers of the submodules.

Method `find_ident`

```
def find_ident(self, name)
```

Searches this module and **all** other public modules for an identifier with name `name` in its list of exported identifiers.

The documentation object corresponding to the identifier is returned. If one cannot be found, then an instance of `External` is returned populated with the given identifier.

Method `functions`

```
def functions(self)
```

Returns all documented module-level functions in the module sorted alphabetically as a list of `Function`.

Method `html`

```
def html(self, external_links=False, link_prefix='', source=True, minify=True,
**kwargs)
```

Returns the documentation for this module as self-contained HTML.

If `minify` is `True`, the resulting HTML is minified.

For explanation of other arguments, see `html()`.

`kwargs` is passed to the `mako` render function.

Method `submodules`

```
def submodules(self)
```

Returns all documented sub-modules of the module sorted alphabetically as a list of `Module`.

Method `text`

```
def text(self, **kwargs)
```

Returns the documentation for this module as plain text.

Method `variables`

```
def variables(self)
```

Returns all documented module-level variables in the module sorted alphabetically as a list of `Variable`.

Class `Variable`

Representation of a variable's documentation. This includes module, class, and instance variables.

Ancestors (in MRO)

- pdoc.__init__.Doc

Instance variables

Variable `cls`

The `Class` object if this is a class or instance variable. If not, this is None.

Variable `instance_var`

True if variable is some class' instance variable (as opposed to class variable).

Variable `qualname`

Module-relative "qualified" name of this documentation object, used for show (e.g. Doc.qualname).

Variable `refname`

Reference name of this documentation object, usually its fully qualified path (e.g. pdoc.Doc.refname). Every documentation object provides this property.

Methods

Method `__init__`

```
def __init__(self, name, module, docstring, *, obj=None, cls=None, instance_var=False)
```

Same as `Doc.__init__()`, except `cls` should be provided as a `Class` object when this is a class or instance variable.
