enum$_{tools}$

Release 0.6.1

Dominic Davis-Foster

Oct 17, 2020

CONTENTS

Tools to expand Python's enum module.

Docs	
Tests	
PyPI	
Anaconda	
Activity	
Other	

This library provides the following:

1. A decorator to add docstrings to `Enum` members from a comment at the end of the line.
2. A `Sphinx` extension to document Enums better than `autoclass` can currently.
3. Additional `Enum` classes with different functionality.

CHAPTER

ONE

INSTALLATION

from PyPI

```
$ python3 -m pip install enum_tools --user
```

from Anaconda

First add the required channels

```
$ conda config --add channels https://conda.anaconda.org/domdfcoding
$ conda config --add channels https://conda.anaconda.org/conda-forge
```

Then install

```
$ conda install enum_tools
```

from GitHub

```
$ python3 -m pip install git+https://github.com/domdfcoding/enum_tools@master --user
```

1.1 `autoenum` Sphinx Extension

.. autoenum::

.. autoflag::
Used to document `Enums` and `Flags` respectively.

The directives support the same options as `autoclass`, but with a few changes to the behaviour:

- Enum members are always shown regardless of whether they are documented or not.
- Enum members are grouped separately from methods.

The docstrings of the Enum members are taken from their `__doc__` attributes. This can be set during initialisation of the enum (see an example here), with the *`DocumentedEnum`* class, or with the *`document_enum()`* decorator.

See https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html for further details.

:py:enum:mem:
:py:enum:member:
:py:flag:mem:
:py:flag:member:
Provides cross-references to Enum/Flag members.

Unlike a standard `:class:` or `:enum:` xref the default behaviour of the ~ prefix is to show both the Enum's name and the member's name, e.g.

```
:py:enum:mem:`~enum_tools.demo.StatusFlags.Running`
```

StatusFlags.Running

The original behaviour can be restored by using the + prefix, e.g.

```
:py:enum:mem:`+enum_tools.demo.StatusFlags.Running`
```

Running

Enable `enum_tools.autoenum` by adding the following to the `extensions` variable in your `conf.py`:

```
extensions = [
    ...
    'enum_tools.autoenum',
    ]
```

For more information see https://www.sphinx-doc.org/en/master/usage/extensions/index.html#third-party-extensions .

1.1.1 Demo

These two have been created with `automodule`.

```
.. automodule:: enum_tools.demo
    :members:
```

enum NoMethods(*value*)

Bases: `enum.IntEnum`

An enumeration of people without any methods.

Member Type `int`

Valid values are as follows:

Bob = <NoMethods.Bob: 1>

A person called Bob

Alice = <NoMethods.Alice: 2>

A person called Alice

Carol = <NoMethods.Carol: 3>

A person called Carol

enum People(*value*)

Bases: `enum.IntEnum`

An enumeration of people.

Member Type `int`

Valid values are as follows:

Bob = <People.Bob: 1>

A person called Bob

Alice = <People.Alice: 2>

A person called Alice

Carol = <People.Carol: 3>
A person called Carol

The Enum and its members also have the following methods:

classmethod iter_values()
Iterate over the values of the Enum.

classmethod as_list()
Return the Enum's members as a list.

Return type List

This one has been created with autoenum.

```
.. autoenum:: enum_tools.demo.People
    :members:
```

enum People(*value*)
Bases: enum.IntEnum

An enumeration of people.

Member Type int

Valid values are as follows:

Bob = <People.Bob: 1>
A person called Bob

Alice = <People.Alice: 2>
A person called Alice

Carol = <People.Carol: 3>
A person called Carol

The Enum and its members also have the following methods:

classmethod iter_values()
Iterate over the values of the Enum.

classmethod as_list()
Return the Enum's members as a list.

Return type List

If members don't have their own docstrings no docstring is shown:

```
.. autoenum:: enum_tools.demo.NoMemberDoc
    :members:
```

enum NoMemberDoc(*value*)
Bases: enum.IntEnum

An enumeration of people without any member docstrings.

Member Type int

Valid values are as follows:

Bob = <NoMemberDoc.Bob: 1>

Alice = <NoMemberDoc.Alice: 2>

Carol = <NoMemberDoc.Carol: 3>

enum.Flags **can also be documented:**

```
.. autoflag:: enum_tools.demo.StatusFlags
    :members:
```

flag StatusFlags(*value*)

Bases: enum.IntFlag

An enumeration of status codes.

Member Type int

Valid values are as follows:

Running = <StatusFlags.Running: 1>

The system is running.

Stopped = <StatusFlags.Stopped: 2>

The system has stopped.

Error = <StatusFlags.Error: 4>

An error has occurred.

The Flag and its members also have the following methods:

has_errored()

Returns whether the operation has errored.

Return type bool

1.2 enum_tools.autoenum

A Sphinx directive for documenting Enums in Python.

Provides the *autoenum* directive for documenting Enums, and *autoflag* for documenting Flags.

It behaves much like autoclass and autofunction.

See also https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html.

Changed in version 0.4.0: Added *PyEnumXRefRole* and *py:enum:mem*.

Classes:

EnumDocumenter(*args)	Sphinx autodoc Documenter for documenting Enums.
EnumMemberDocumenter(directive, name[, indent])	Sphinx autodoc Documenter for documenting Enum members.
FlagDocumenter(*args)	Sphinx autodoc Documenter for documenting Flags.
PyEnumXRefRole([fix_parens, lowercase, ...])	XRefRole for Enum/Flag members.

Functions:

setup(app)	Setup Sphinx Extension.

class EnumDocumenter(**args*)

Bases: ClassDocumenter

Sphinx autodoc `Documenter` for documenting `Enum`s.

Methods:

can_document_member(member, membername, ...)	Called to see if a member can be documented by this documenter.
document_members([all_members])	Generate reST for member documentation.
generate([more_content, real_modname, ...])	Generate reST for the object given by *self.name*, and possibly for its members.

classmethod can_document_member(*member*, *membername*, *isattr*, *parent*)
Called to see if a member can be documented by this documenter.

Parameters

- **member** (`Any`)
- **membername** (`str`)
- **isattr** (`bool`)
- **parent** (`Any`)

Return type `bool`

document_members(*all_members=False*)
Generate reST for member documentation.

Parameters **all_members** (`bool`) – If `True`, document all members, otherwise document those given by else those given by `self.options.members`. Default `False`.

generate(*more_content=None*, *real_modname=None*, *check_module=False*, *all_members=False*)
Generate reST for the object given by *self.name*, and possibly for its members.

Parameters

- **more_content** (`Optional`[`Any`]) – Additional content to include in the reST output. Default `None`.
- **real_modname** (`Optional`[`str`]) – Module name to use to find attribute documentation. Default `None`.
- **check_module** (`bool`) – If `True`, only generate if the object is defined in the module name it is imported from. Default `False`.
- **all_members** (`bool`) – If `True`, document all members. Default `False`.

class EnumMemberDocumenter(*directive*, *name*, *indent=''*)
Bases: `AttributeDocumenter`

Sphinx autodoc `Documenter` for documenting `Enum` members.

Methods:

generate([more_content, real_modname, ...])	Generate reST for the object given by `self.name`, and possibly for its members.
import_object([raiseerror])	Import the object given by `self.modname` and `self.objpath` and set it as `self.object`.

generate(*more_content=None*, *real_modname=None*, *check_module=False*, *all_members=False*)
Generate reST for the object given by `self.name`, and possibly for its members.

Parameters

- **more_content** (Optional[Any]) – Additional content to include in the reST output. Default None.
- **real_modname** (Optional[str]) – Module name to use to find attribute documentation. Default None.
- **check_module** (bool) – If True, only generate if the object is defined in the module name it is imported from. Default False.
- **all_members** (bool) – If True, document all members. Default False.

import_object(*raiseerror=False*)

Import the object given by self.modname and self.objpath and set it as self.object.

Parameters **raiseerror** (bool) – Default False.

Return type Any

Returns True if successful, False if an error occurred.

class FlagDocumenter(**args*)

Bases: *EnumDocumenter*

Sphinx autodoc Documenter for documenting Flags.

Methods:

can_document_member(member, membername, ...)	Called to see if a member can be documented by this documenter.

classmethod can_document_member(*member*, *membername*, *isattr*, *parent*)

Called to see if a member can be documented by this documenter.

Parameters

- **member** (Any)
- **membername** (str)
- **isattr** (bool)
- **parent** (Any)

Return type bool

class PyEnumXRefRole(*fix_parens=False*, *lowercase=False*, *nodeclass=None*, *innernodeclass=None*, *warn_dangling=False*)

Bases: PyXRefRole

XRefRole for Enum/Flag members.

New in version 0.4.0.

Methods:

process_link(env, refnode, ...)	Called after parsing title and target text, and creating the reference node (given in *refnode*).

process_link(*env*, *refnode*, *has_explicit_title*, *title*, *target*)

Called after parsing title and target text, and creating the reference node (given in *refnode*). This method can alter the reference node and must return a new (or the same) (title, target) tuple.

Return type Tuple[str, str]

setup(*app*)
Setup Sphinx Extension.

Parameters **app** (Sphinx)

Return type Dict[str, Any]

1.3 enum_tools.custom_enums

Custom subclasses of enum.Enum and enum.Flag.

Classes:

AutoNumberEnum(value)	Enum that automatically assigns increasing values to members.
DuplicateFreeEnum(*args)	Enum that disallows duplicated member names.
IntEnum(value)	Enum where members are also (and must be) ints.
IterableFlag(value)	enum.Flag with support for iterating over members and member combinations.
IterableIntFlag(value)	enum.IntFlag with support for iterating over members and member combinations.
MemberDirEnum(value)	Enum which includes attributes as well as methods.
OrderedEnum(value)	Enum that adds ordering based on the values of its members.
StrEnum(value)	Enum where members are also (and must be) strings.

enum AutoNumberEnum(*value*)
Bases: enum.Enum

Enum that automatically assigns increasing values to members.

enum DuplicateFreeEnum(*value*)
Bases: enum.Enum

Enum that disallows duplicated member names.

enum IntEnum(*value*)
Bases: int, enum.Enum

Enum where members are also (and must be) ints.

Member Type int

flag IterableFlag(*value*)
Bases: enum.Flag

enum.Flag with support for iterating over members and member combinations.

This functionality was added to Python 3.10's enum module in #22221#22221.

New in version 0.5.0.

flag IterableIntFlag(*value*)
Bases: enum.IntFlag

enum.IntFlag with support for iterating over members and member combinations.

This functionality was added to Python 3.10's enum module in #22221#22221.

New in version 0.5.0.

Member Type `int`

enum MemberDirEnum(*value*)
Bases: `enum.Enum`

`Enum` which includes attributes as well as methods.

This will be part of the `enum` module starting with Python 3.10.

See also:

Pull request #19219#19219 by Angelin BOOZ, which added this to CPython.

New in version 0.6.0.

enum OrderedEnum(*value*)
Bases: `enum.Enum`

`Enum` that adds ordering based on the values of its members.

The `Enum` and its members have the following methods:

__ge__(*other*)
Return self>=value.

Return type `bool`

__gt__(*other*)
Return self>value.

Return type `bool`

__le__(*other*)
Return self<=value.

Return type `bool`

__lt__(*other*)
Return self<value.

Return type `bool`

enum StrEnum(*value*)
Bases: `str`, `enum.Enum`

`Enum` where members are also (and must be) strings.

Member Type `str`

1.4 enum_tools.documentation

1.4.1 Core Functionality

Decorators to add docstrings to enum members from comments.

Classes:

DocumentedEnum(value)	An enum where docstrings are automatically added to members from comments starting with `doc:`.

Functions:

document_enum(an_enum)	Document all members of an enum by adding a comment to the end of each line that starts with `doc:`.
document_member(enum_member)	Document a member of an enum by adding a comment to the end of the line that starts with `doc:`.

enum DocumentedEnum(*value*)
Bases: `enum.Enum`

An enum where docstrings are automatically added to members from comments starting with `doc:`.

@**document_enum**(*an_enum*)
Document all members of an enum by adding a comment to the end of each line that starts with `doc:`.

Parameters **an_enum** (`EnumMeta`) – An `Enum` subclass

Return type `EnumMeta`

document_member(*enum_member*)
Document a member of an enum by adding a comment to the end of the line that starts with `doc:`.

Parameters **enum_member** (`Enum`) – A member of an `Enum` subclass

1.4.2 Utilities

Functions:

get_base_indent(base_indent, all_tokens, indent)	Determine the base level of indentation (i.e.
get_dedented_line(line)	Returns the line without indentation, and the amount of indentation.
get_tokens(line)	Returns a list ot tokens generated from the given Python code.
parse_tokens(all_tokens)	Parse the tokens representing a line of code to identify Enum members and `doc:` comments.

get_base_indent(*base_indent*, *all_tokens*, *indent*)
Determine the base level of indentation (i.e. one level of indentation in from the `c` of `class`).

Parameters

- **base_indent** (`Optional[int]`) – The current base level of indentation
- **all_tokens**
- **indent** (*int*) – The current level of indentation

Return type `Optional[int]`

Returns The base level of indentation

get_dedented_line(*line*)
Returns the line without indentation, and the amount of indentation.

Parameters **line** (*str*) – A line of Python source code

Return type `Tuple[int, str]`

Returns

get_tokens(*line*)
Returns a list ot tokens generated from the given Python code.

Parameters **line** (str) – Line of Python code to tokenise.

Return type List[Tuple]

parse_tokens(*all_tokens*)
Parse the tokens representing a line of code to identify Enum members and `doc:` comments.

Parameters **all_tokens**

Returns A list of the Enum members' names, and the docstring for them.

1.5 enum_tools.utils

General utility functions.

Classes:

HasMRO	typing.Protocol for classes that have a Method Resolution Order magic method.

Functions:

get_base_object(enum)	Returns the object type of the enum's members.
is_enum(obj)	Returns True if obj is an enum.Enum.
is_enum_member(obj)	Returns True if obj is an enum.Enum member.
is_flag(obj)	Returns True if obj is an enum.Flag.

protocol HasMRO
Bases: typing.Protocol

typing.Protocol for classes that have a Method Resolution Order magic method.

This protocol is runtime checkable.

Classes that implement this protocol must have the following methods / attributes:

__mro__ = (<class 'enum_tools.utils.HasMRO'>, <class 'typing.Protocol'>, <class 'typin

get_base_object(*enum*)
Returns the object type of the enum's members.

If the members are of indeterminate type object is returned.

Return type Type

is_enum(*obj*)
Returns True if obj is an enum.Enum.

Return type bool

is_enum_member(*obj*)
Returns True if obj is an enum.Enum member.

Return type bool

is_flag(*obj*)
Returns True if obj is an enum.Flag.

Return type bool

1.6 Overview

enum_tools uses tox to automate testing and packaging, and pre-commit to maintain code quality.

Install pre-commit with pip and install the git hook:

```
$ python -m pip install pre-commit
$ pre-commit install
```

1.7 Coding style

Yapf is used for code formatting, and isort is used to sort imports.

yapf and isort can be run manually via pre-commit:

```
$ pre-commit run yapf -a
$ pre-commit run isort -a
```

The complete autoformatting suite can be run with pre-commit:

```
$ pre-commit run -a
```

1.8 Automated tests

Tests are run with tox and pytest. To run tests for a specific Python version, such as Python 3.6, run:

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

1.9 Type Annotations

Type annotations are checked using mypy. Run mypy using tox:

```
$ tox -e mypy
```

1.10 Build documentation locally

The documentation is powered by Sphinx. A local copy of the documentation can be built with tox:

```
$ tox -e docs
```

1.11 Downloading source code

The `enum_tools` source code is available on GitHub, and can be accessed from the following URL: https://github.com/domdfcoding/enum_tools"

If you have `git` installed, you can clone the repository with the following command:

```
$ git clone https://github.com/domdfcoding/enum_tools"
> Cloning into 'enum_tools'...
> remote: Enumerating objects: 47, done.
> remote: Counting objects: 100% (47/47), done.
> remote: Compressing objects: 100% (41/41), done.
> remote: Total 173 (delta 16), reused 17 (delta 6), pack-reused 126
> Receiving objects: 100% (173/173), 126.56 KiB | 678.00 KiB/s, done.
> Resolving deltas: 100% (66/66), done.
```

Alternatively, the code can be downloaded in a 'zip' file by clicking:
Clone or download –> Download Zip

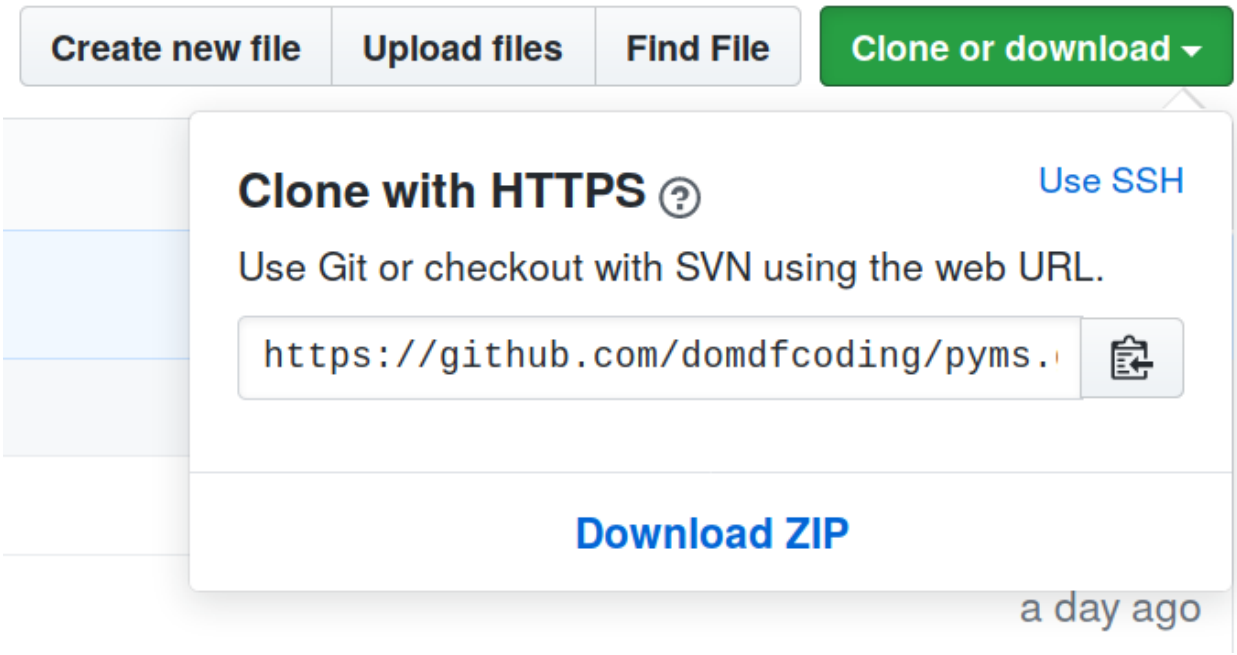

Fig. 1: Downloading a 'zip' file of the source code

1.11.1 Building from source

The recommended way to build `enum_tools` is to use tox:

```
$ tox -e build
```

The source and wheel distributions will be in the directory `dist`.

If you wish, you may also use pep517.build or another **PEP 517**-compatible build tool.

View the Function Index or browse the Source Code.

Browse the GitHub Repository

CHAPTER

TWO

FURTHER READING

1. https://docs.python.org/3/library/enum.html
2. Is it possible to define a class constant inside an Enum?
3. Enums with Attributes
4. When should I subclass EnumMeta instead of Enum?

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

M

P

R

S