attr-utils

Release 1.0.0

Utilities to augment attrs.

Dominic Davis-Foster

Apr 04, 2024

Contents

Chapter

ONE

Installation

1.1 from PyPI

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

1.2 from Anaconda

First add the required channels

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

Then install

```
$ conda install attr_utils
```

1.3 from GitHub

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

Attention: In v0.6.0 and above the *pprinter* module requires the `pprint` extra to be installed:

```
python -m pip install attr-utils[pprint]
```

Chapter

TWO

Overview

attr_utils provides both utility functions and two Sphinx extensions: *attr_utils.annotations* and *attr_utils.autoattrs*.

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

```
extensions = [
    ...
    'sphinx.ext.autodoc',
    'sphinx_toolbox.more_autodoc.typehints',
    'attr_utils.annotations',
    'attr_utils.autoattrs',
    ]
```

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

Chapter

THREE

Demo

This example shows the output from `attr_utils.annotations` and `attr_utils.autoattrs`.

```python
"""
This example is based on real code.
"""

# stdlib
from typing import Any, Dict, List, Optional, Sequence, Tuple, Union, overload

# 3rd party
import attr
from domdf_python_tools.utils import strtobool

# this package
from attr_utils.annotations import attrib
from attr_utils.pprinter import pretty_repr
from attr_utils.serialise import serde


@pretty_repr
@serde
@attr.s(slots=True)
class Device:
    """
    Represents a device in an :class:`~.AcqMethod`.
    """

    #: The ID of the device
    device_id: str = attr.ib(converter=str)

    #: The display name for the device.
    display_name: str = attr.ib(converter=str)

    rc_device: bool = attr.ib(converter=strtobool)
    """
    Flag to indicate the device is an RC Device.
    If :py:obj:`False` the device is an SCIC.
    """

    #: List of key: value mappings for configuration options.
    configuration: List[Dict[str, Any]] = attr.ib(converter=list, factory=list)

    #: Alternative form of ``configuration``.
    configuration2: Tuple[Dict[str, Any]] = attr.ib(
            converter=tuple,
            default=attr.Factory(tuple),
            )

```

(continues on next page)

(continued from previous page)

```
    #: Alternative form of ``configuration``.
    configuration3: List[Dict[str, Any]] = attr.ib(
            converter=list,
            default=attr.Factory(list),
            metadata={"annotation": Sequence[Dict[str, Any]]},
            )

    #: Alternative form of ``configuration``.
    configuration4: List[Dict[str, Any]] = attrib(
            converter=list,
            factory=list,
            annotation=Sequence[Dict[str, Any]],
            )

    @overload
    def __getitem__(self, item: int) -> str: ...

    @overload
    def __getitem__(self, item: slice) -> List[str]: ...

    def __getitem__(self, item: Union[int, slice]) -> Union[str, List[str]]:
        """
        Return the item with the given index.

        :param item:

        :rtype:

        .. versionadded:: 1.2.3
        """


@attr.s(init=False)
class Connector:
    """
    Represents an electrical connector.

    :param name: The name of the connector.
    :param n_pins: The number if pins. For common connectors this is inferred from␣
↪the name.
    :param right_angle: Whether this is a right angle connector.
    """

    #: The name of the connector
    name: str = attr.ib(converter=str)

    #: The number of pins
    n_pins: int = attr.ib(converter=int)

    def __init__(self, name: str, n_pins: Optional[int] = None, right_angle: bool =␣
↪False):
        if name == "DA-15":
            n_pins = 15
        elif name == "DB-25":
            n_pins = 25
        elif name == "DE-15":
            n_pins = 15
```

(continues on next page)

(continued from previous page)

```

        self.__attrs_init__(name, n_pins)
```

```
.. autoattrs:: demo.Device
    :autosummary:
```

class Device(*device_id*, *display_name*, *rc_device*, *configuration=[]*, *configuration2=()*, *configuration3=[]*, *configuration4=[]*)

Bases: `object`

Represents a device in an `AcqMethod`.

Parameters

- **device_id** (`str`) – The ID of the device
- **display_name** (`str`) – The display name for the device.
- **rc_device** (`Union[str, int]`) – Flag to indicate the device is an RC Device. If `False` the device is an SCIC.
- **configuration** (`List[Dict[str, Any]]`) – List of key: value mappings for configuration options. Default `[]`.
- **configuration2** (`Tuple[Dict[str, Any]]`) – Alternative form of `configuration`. Default `()`.
- **configuration3** (`Sequence[Dict[str, Any]]`) – Alternative form of `configuration`. Default `[]`.
- **configuration4** (`Sequence[Dict[str, Any]]`) – Alternative form of `configuration`. Default `[]`.

Methods:

__eq__(other)	Return `self == other`.
__ge__(other)	Return `self >= other`.
__getitem__(item)	Return the item with the given index.
__getstate__()	Used for pickling.
__gt__(other)	Return `self > other`.
__le__(other)	Return `self <= other`.
__lt__(other)	Return `self < other`.
__ne__(other)	Return `self != other`.
__repr__()	Return a string representation of the *`Device`*.
__setstate__(state)	Used for pickling.
from_dict(d)	Construct an instance of *`Device`* from a dictionary.
to_dict([convert_values])	Returns a dictionary containing the contents of the *`Device`* object.

Attributes:

configuration	List of key: value mappings for configuration options.
configuration2	Alternative form of `configuration`.

continues on next page

Table 2 – continued from previous page

configuration3	Alternative form of `configuration`.
configuration4	Alternative form of `configuration`.
device_id	The ID of the device
display_name	The display name for the device.
rc_device	Flag to indicate the device is an RC Device.

__eq__(*other*)

Return `self == other`.

Return type `bool`

__ge__(*other*)

Return `self >= other`.

Return type `bool`

__getitem__(*item*)

Return the item with the given index.

Parameters **item** (`Union[int, slice]`)

Return type `Union[str, List[str]]`

Overloads

- *__getitem__*(**item**: `int`) -> `str`
- *__getitem__*(**item**: `slice`) -> `List[str]`

New in version 1.2.3.

__getstate__()

Used for pickling.

Automatically created by attrs.

__gt__(*other*)

Return `self > other`.

Return type `bool`

__le__(*other*)

Return `self <= other`.

Return type `bool`

__lt__(*other*)

Return `self < other`.

Return type `bool`

__ne__(*other*)

Return `self != other`.

Return type `bool`

__repr__()

Return a string representation of the *Device*.

Return type str

__setstate__(*state*)

Used for pickling.

Automatically created by attrs.

configuration

Type: List[Dict[str, Any]]

List of key: value mappings for configuration options.

configuration2

Type: Tuple[Dict[str, Any]]

Alternative form of configuration.

configuration3

Type: List[Dict[str, Any]]

Alternative form of configuration.

configuration4

Type: List[Dict[str, Any]]

Alternative form of configuration.

device_id

Type: str

The ID of the device

display_name

Type: str

The display name for the device.

classmethod from_dict(*d*)

Construct an instance of *Device* from a dictionary.

Parameters **d** (Mapping[str, Any]) – The dictionary.

rc_device

Type: bool

Flag to indicate the device is an RC Device. If False the device is an SCIC.

to_dict(*convert_values=False*)

Returns a dictionary containing the contents of the *Device* object.

Parameters **convert_values** (bool) – Recursively convert values into dictionaries, lists etc. as appropriate. Default False.

Return type MutableMapping[str, Any]

Chapter

FOUR

attr_utils.annotations

Add type annotations to the `__init__` of an attrs class.

Since python-attrs/attrs#363 attrs has populated the `__init__.__annotations__` based on the types of attributes. However, annotations were deliberately omitted when converter functions were used. This module attempts to generate the annotations for use in Sphinx documentation, even when converter functions *are* used, based on the following assumptions:

- If the converter function is a Python `type`, such as `str`, `int`, or `list`, the type annotation will be that type. If the converter and the type annotation refer to the same type (e.g. `list` and `typing.List`) the type annotation will be used.
- If the converter function has an annotation for its first argument, that annotation is used.
- If the converter function is not annotated, the type of the attribute will be used.

The annotation can also be provided via the `'annotation'` key in the metadata dict. If you prefer you can instead provide this as a keyword argument to *`attrib()`* which will construct the metadata dict and call `attr.ib()` for you.

Changed in version 0.2.0: Improved support for container types.

> **Attention:** Due to changes in the `typing` module *`annotations`* is only officially supported on Python 3.7 and above.

4.1 Examples

Library Usage:

```
def my_converter(arg: List[Dict[str, Any]]):
    return arg


def untyped_converter(arg):
    return arg


@attr.s
class SomeClass:
    a_string: str = attr.ib(converter=str)
    custom_converter: Any = attr.ib(converter=my_converter)
    untyped: Tuple[str, int, float] = attr.ib(converter=untyped_converter)
    annotated: List[str] = attr.ib(
        converter=list,
        metadata={"annotation": Sequence[str]},
    )
```

(continues on next page)

(continued from previous page)

```

add_attrs_annotations(SomeClass)

print(SomeClass.__init__.__annotations__)
# {
#    'return': None,
#    'a_string': <class 'str'>,
#    'custom_converter': typing.List[typing.Dict[str, typing.Any]],
#    'untyped': typing.Tuple[str, int, float],
#    }
```

Sphinx documentation:

```
@attr.s
class AttrsClass:
    """
    Example of using :func:`~.add_init_annotations` for attrs_ classes with Sphinx␣
↪documentation.

    .. _attrs: https://www.attrs.org/en/stable/

    :param name: The name of the person.
    :param age: The age of the person.
    :param occupations: The occupation(s) of the person.
    """

    name: str = attr.ib(converter=str)
    age: int = attr.ib(converter=int)
    occupations: List[str] = attr.ib(converter=parse_occupations)
```

The `parse_occupations` function looks like:

```
def parse_occupations(occupations: Iterable[str]) -> Iterable[str]:  # pragma: no␣
↪cover

    if isinstance(occupations, str):
        return [x.strip() for x in occupations.split(',')]
    else:
        return [str(x) for x in occupations]
```

The Sphinx output looks like:

class AttrsClass(*name*, *age*, *occupations*)

Bases: `object`

Example of using `add_init_annotations()` for attrs classes with Sphinx documentation.

Parameters

- **name** (`str`) – The name of the person.
- **age** (`int`) – The age of the person.
- **occupations** (`Iterable[str]`) – The occupation(s) of the person.

4.2 API Reference

Functions:

attrib([default, validator, repr, hash, ...])	Wrapper around `attr.ib()` which supports the `annotation` keyword argument for use by *add_init_annotations()*.
add_init_annotations(obj)	Add type annotations to the `__init__` method of an attrs class.
attr_docstring_hook(obj)	Hook for `sphinx_toolbox.more_autodoc.typehints` to add annotations to the `__init__` method of attrs classes.
setup(app)	Sphinx extension to populate `__init__.__annotations__` for attrs classes.

Data:

_A	Invariant `TypeVar` bound to `typing.Any`.
_C	Invariant `TypeVar` bound to `typing.Callable`.

attrib(*default=NOTHING*, *validator=None*, *repr=True*, *hash=None*, *init=True*, *metadata=None*, *annotation=NOTHING*, *converter=None*, *factory=None*, *kw_only=False*, *eq=None*, *order=None*, ***kwargs*)

Wrapper around `attr.ib()` which supports the `annotation` keyword argument for use by *add_init_annotations()*.

New in version 0.2.0.

Parameters

- **default** – Default `NOTHING`.
- **validator** – Default `None`.
- **repr** (`bool`) – Default `True`.
- **hash** – Default `None`.
- **init** – Default `True`.
- **metadata** – Default `None`.
- **annotation** (`Union[Type, object]`) – The type to add to `__init__.__annotations__`, if different to that the type taken as input to the converter function or the type hint of the attribute. Default `NOTHING`.
- **converter** – Default `None`.
- **factory** – Default `None`.
- **kw_only** (`bool`) – Default `False`.
- **eq** – Default `None`.
- **order** – Default `None`.

See the documentation for `attr.ib()` for descriptions of the other arguments.

_A = TypeVar(_A, bound=typing.Any)

Type: `TypeVar`

Invariant `TypeVar` bound to `typing.Any`.

_C = TypeVar(_C, bound=typing.Callable)

Type: TypeVar

Invariant TypeVar bound to typing.Callable.

add_init_annotations(*obj*)

Add type annotations to the __init__ method of an attrs class.

Return type ~_C

attr_docstring_hook(*obj*)

Hook for sphinx_toolbox.more_autodoc.typehints to add annotations to the __init__ method of attrs classes.

Parameters **obj** (~_A) – The object being documented.

Return type ~_A

setup(*app*)

Sphinx extension to populate __init__.__annotations__ for attrs classes.

Parameters **app** (Sphinx)

Return type SphinxExtMetadata

Chapter

FIVE

attr_utils.autoattrs

Sphinx directive for documenting attrs classes.

New in version 0.1.0.

Attention: Due to changes in the `typing` module *`autoattrs`* is only officially supported on Python 3.7 and above.

Attention: This module has the following additional requirements:

```
sphinx<7,>=3.2.0
sphinx-toolbox>=3.3.0
```

These can be installed as follows:

```
$ python -m pip install attr-utils[sphinx]
```

`.. autoattrs::`

Autodoc directive to document an attrs class.

It behaves much like `autoclass`. It can be used directly or as part of `automodule`.

Docstrings for parameters in `__init__` can be given in the class docstring or alongside each attribute (see `autoattribute` for the syntax). The second option is recommended as it interacts better with other parts of autodoc. However, the class docstring can be used to override the description for a given parameter.

`:no-init-attribs: (flag)`

Excludes attributes taken as arguments in `__init__` from the output, even if they are documented.

This may be useful for simple classes where converter functions aren't used.

This option cannot be used as part of `automodule`.

5.2 API Reference

Classes:

AttrsDocumenter(*args)	Sphinx autodoc `Documenter` for documenting attrs classes.

Functions:

setup(app)	Setup *attr_utils.autoattrs*.

class AttrsDocumenter(**args*)

Bases: `PatchedAutoSummClassDocumenter`

Sphinx autodoc `Documenter` for documenting attrs classes.

Changed in version 0.3.0:

- Parameters for `__init__` can be documented either in the class docstring or alongside the attribute. The class docstring has priority.
- Added support for autodocsumm.

Methods:

can_document_member(member, membername, ...)	Called to see if a member can be documented by this documenter.
add_content(more_content[, no_docstring])	Add extra content (from docstrings, attribute docs etc.), but not the class docstring.
import_object([raiseerror])	Import the object given by `self.modname` and `self.objpath` and set it as `self.object`.
sort_members(documenters, order)	Sort the given member list and add attribute docstrings to the class docstring.
filter_members(members, want_all)	Filter the list of members to always include init attributes unless the `:no-init-attribs:` flag was given.
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`

add_content (*more_content*, *no_docstring=False*)
Add extra content (from docstrings, attribute docs etc.), but not the class docstring.

Parameters

- **more_content** (Any)
- **no_docstring** (bool) – Default False.

import_object (*raiseerror=False*)
Import the object given by self.modname and self.objpath and set it as self.object.

If the object is an attrs class *attr_utils.docstrings.add_attrs_doc()* will be called.

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

Return type bool

Returns True if successful, False if an error occurred.

sort_members (*documenters*, *order*)
Sort the given member list and add attribute docstrings to the class docstring.

Parameters

- **documenters** (List[Tuple[Documenter, bool]])
- **order** (str)

Return type List[Tuple[Documenter, bool]]

filter_members (*members*, *want_all*)
Filter the list of members to always include init attributes unless the :no-init-attribs: flag was given.

Parameters

- **members** (List[Tuple[str, Any]])
- **want_all** (bool)

Return type List[Tuple[str, Any, bool]]

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.

setup (*app*)
Setup *attr_utils.autoattrs*.

Parameters **app** (Sphinx)

Return type `SphinxExtMetadata`

Chapter

SIX

attr_utils.docstrings

Add better docstrings to attrs generated functions.

Data:

_T	Invariant `TypeVar` bound to `typing.Type`.

Functions:

add_attrs_doc(obj)	Add better docstrings to attrs generated functions.

_T = TypeVar(_T, bound=typing.Type)

Type: `TypeVar`

Invariant `TypeVar` bound to `typing.Type`.

add_attrs_doc(*obj*)

Add better docstrings to attrs generated functions.

Parameters **obj** (*~_T*) – The class to improve the docstrings for.

Return type *~_T*

Chapter

SEVEN

attr_utils.mypy_plugin

Plugin for mypy which adds support for attr_utils.

New in version 0.4.0.

To use this plugin, add the following to your mypy configuration file:

```
[mypy]
plugins=attr_utils.mypy_plugin
```

See https://mypy.readthedocs.io/en/stable/extending_mypy.html#configuring-mypy-to-use-plugins for more information.

Classes:

AttrUtilsPlugin	Plugin for mypy which adds support for `attr_utils`.

Functions:

add_classmethod_to_class(api, cls, name, ...)	Adds a new classmethod to a class definition.
attr_utils_serialise_serde(cls_def_ctx)	Handles *attr_utils.serialise.serde()*.
plugin(version)	Entry point to *attr_utils.mypy_plugin*.

class AttrUtilsPlugin

Bases: `Plugin`

Plugin for mypy which adds support for `attr_utils`.

get_class_decorator_hook(*fullname*)

Allows mypy to handle decorators that add extra methods to classes.

Parameters **fullname** (`str`) – The full name of the decorator.

Return type `Optional[Callable[[ClassDefContext], None]]`

add_classmethod_to_class(*api*, *cls*, *name*, *args*, *return_type*, *cls_type=None*, *tvar_def=None*)

Adds a new classmethod to a class definition.

Parameters

- **api** (`SemanticAnalyzerPluginInterface`)
- **cls** (`ClassDef`)
- **name** (`str`)
- **args** (`List[Argument]`)
- **return_type** (`Type`)

- **cls_type** (Optional[Type]) – Default None.
- **tvar_def** (Optional[Any]) – Default None.

attr_utils_serialise_serde(*cls_def_ctx*)

Handles *attr_utils.serialise.serde()*.

Parameters **cls_def_ctx** (ClassDefContext)

plugin(*version*)

Entry point to *attr_utils.mypy_plugin*.

Parameters **version** (str) – The current mypy version.

Chapter

EIGHT

attr_utils.pprinter

Pretty printing functions.

This module monkeypatches prettyprinter to disable a potentially undesirable behaviour of its singledispatch feature, where deferred types took precedence over resoled types.

It also changes the pretty print output for an `enum.Enum` to be the same as the `Enum`'s `__repr__`.

Attention: This module has the following additional requirement:

```
prettyprinter==0.18.0
```

This can be installed as follows:

```
$ python -m pip install attr-utils[pprint]
```

Classes:

PrettyFormatter	`typing.Protocol` representing the pretty formatting functions decorated by *register_pretty()*.

Data:

_PF	Invariant `TypeVar` bound to *attr_utils.pprinter.PrettyFormatter*.

Functions:

pretty_repr(obj)	Add a pretty-printing `__repr__` function to the decorated attrs class.
register_pretty([type, predicate])	Returns a decorator that registers the decorated function as the pretty printer for instances of `type`.

protocol PrettyFormatter

Bases: `Protocol`

`typing.Protocol` representing the pretty formatting functions decorated by *register_pretty()*.

New in version 0.6.0.

This protocol is runtime checkable.

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

__call__(*value*, *ctx*)

Call the function.

Parameters

- **value** (Any) – The value to pretty print.
- **ctx** (Any) – The context.

Return type str

__non_callable_proto_members__ = {}

Type: set

_PF = TypeVar(_PF, bound=PrettyFormatter)

Type: TypeVar

Invariant TypeVar bound to *attr_utils.pprinter.PrettyFormatter*.

pretty_repr(*obj*)

Add a pretty-printing __repr__ function to the decorated attrs class.

```
>>> import attr
>>> from attr_utils.pprinter import pretty_repr

>>> @pretty_repr
... @attr.s
... class Person(object):
...     name = attr.ib()

>>> repr(Person(name="Bob"))
Person(name='Bob')
```

Parameters **obj** (Type)

register_pretty(*type=None*, *predicate=None*)

Returns a decorator that registers the decorated function as the pretty printer for instances of type.

Parameters

- **type** (Union[Type, str, None]) – The type to register the pretty printer for, or a str to indicate the module and name, e.g. 'collections.Counter'. Default None.
- **predicate** (Optional[Callable[[Any], bool]]) – A predicate function that takes one argument and returns a boolean indicating if the value should be handled by the registered pretty printer. Default None.

Only one of type and predicate may be supplied, and therefore predicate will only be called for unregistered types.

Return type Callable[[~_PF], ~_PF]

Here's an example of the pretty printer for collections.OrderedDict:

```
from collections import OrderedDict
from attr_utils.pprinter import register_pretty
from prettyprinter import pretty_call

@register_pretty(OrderedDict)
def pretty_orderreddict(value, ctx):
    return pretty_call(ctx, OrderedDict, list(value.items()))
```

Chapter

NINE

attr_utils.serialise

Add serialisation and deserialisation capability to attrs classes.

Based on attrs-serde.

9.1 Example usage

```
>>> import attr
>>> from attr_utils.serialise import serde

>>> person_dict = {"contact": {"personal": {"name": "John"}, "phone": "555-112233"}}

>>> name_path = ["contact", "personal", "name"]
>>> phone_path = ["contact", "phone"]

>>> @serde
... @attr.s
... class Person(object):
...     name = attr.ib(metadata={"to": name_path, "from": name_path})
...     phone = attr.ib(metadata={"to": phone_path, "from": phone_path})

>>> p = Person.from_dict(person_dict)
Person(name=John phone=555-112233)

>>> p.to_dict
{"contact": {"personal": {"name": "John"}, "phone": "555-112233"}}
```

9.2 API Reference

serde(*cls=None*, *from_key='from'*, *to_key='to'*)

Decorator to add serialisation and deserialisation capabilities to attrs classes.

The keys used in the dictionary output, and used when creating the class from a dictionary, can be controlled using the `metadata` argument in `attr.ib()`:

```
from attr_utils.serialize import serde
import attr

@serde
@attr.s
class Person(object):
    name = attr.ib(metadata={"to": name_path, "from": name_path})
    phone = attr.ib(metadata={"to": phone_path, "from": phone_path})
```

The names of the keys given in the `metadata` argument can be controlled with the `from_key` and `to_key` arguments:

```
from attr_utils.serialize import serde
import attr

@serde(from_key="get", to_key="set")
@attr.s
class Person(object):
    name = attr.ib(metadata={"get": name_path, "set": name_path})
    phone = attr.ib(metadata={"get": phone_path, "set": phone_path})
```

This may be required when using other extensions to attrs.

Parameters

- **cls** (Optional[Type[AttrsClass]]) – The attrs class to add the methods to. Default None.
- **from_key** (str) – Default 'from'.
- **to_key** (str) – Default 'to'.

Return type Union[Type[AttrsClass], Callable[[Type[AttrsClass]], Type[AttrsClass]]]

Overloads

- *serde*(**cls**: Type, **from_key** = ..., **to_key** = ...) -> Type[AttrsClass]
- *serde*(**cls**: None = None, **from_key** = ..., **to_key** = ...) -> Callable[[Type[AttrsClass]], Type[AttrsClass]]

Classes decorated with *@serde* will have two new methods added:

classmethod from_dict(*d*)

Construct an instance of the class from a dictionary.

Parameters **d** (Mapping[str, Any]) – The dictionary.

to_dict(convert_values=False):

Returns a dictionary containing the contents of the class.

Parameters **convert_values** (bool) – Recurse into other attrs classes, and convert tuples, sets etc. into lists. This may be required to later construct a new class from the dictionary if the class uses complex converter functions.

Return type MutableMapping[str, Any]

Changed in version 0.5.0: By default values are left unchanged. In version 0.4.0 these were converted to basic Python types, which may be undesirable. The original behaviour can be restored using the `convert_values` parameter.

Chapter

TEN

Contributing

attr_utils 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
```

10.1 Coding style

formate is used for code formatting.

It can be run manually via pre-commit:

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

Or, to run the complete autoformatting suite:

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

10.2 Automated tests

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

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

10.3 Type Annotations

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

```
$ tox -e mypy
```

10.4 Build documentation locally

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

```
$ tox -e docs
```

Chapter

ELEVEN

Downloading source code

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

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

```
$ git clone https://github.com/domdfcoding/attr_utils
```

```
Cloning into 'attr_utils'...
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

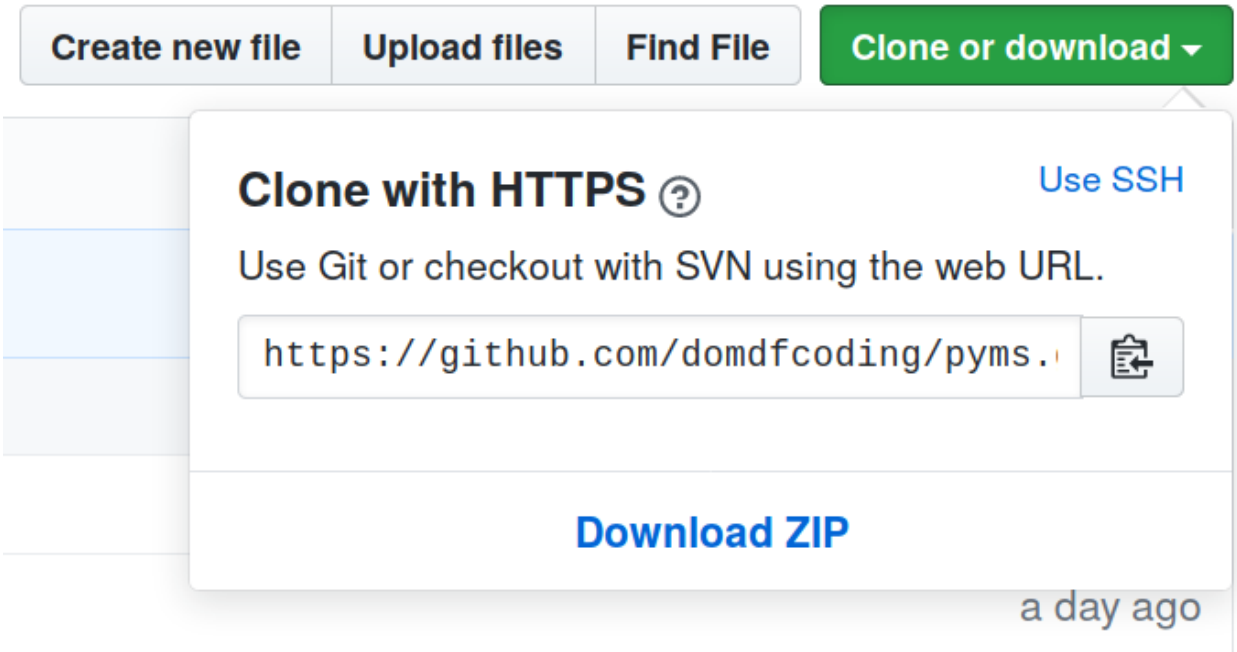

Fig. 1: Downloading a 'zip' file of the source code

11.1 Building from source

The recommended way to build `attr_utils` 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.

Chapter

TWELVE

License

`attr_utils` is licensed under the MIT License

A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Permissions

- Commercial use – The licensed material and derivatives may be used for commercial purposes.
- Modification – The licensed material may be modified.
- Distribution – The licensed material may be distributed.
- Private use – The licensed material may be used and modified in private.

Conditions

- License and copyright notice – A copy of the license and copyright notice must be included with the licensed material.

Limitations

- Liability – This license includes a limitation of liability.
- Warranty – This license explicitly states that it does NOT provide any warranty.

See more information on choosealicense.com ⇒

Python Module Index

a

Index

Symbols

A

C

D

F

G

I

M

P

R

S

T