uttrs

Release 0.5

JBCabral and QuatroPe

Jul 14, 2021

CONTENTS

`uttrs` seeks to interoperate Classes definided using attrs and *astropy units* in a simple manner.

uttrs is mainly three functions:

- `uttr.ib` which generates attributes sensitive to units.
- `uttr.array_accessor` which allows access to attributes linked to units, and transform them into numpy arrays.
- `uttr.s` a class decorator to automatically add the `array_accessor`.

CHAPTER

ONE

CODE AND ISSUES

The entire source code of is hosted in GitHub https://github.com/quatrope/uttrs/

CHAPTER

TWO

LICENSE

Uttrs is under The BSD-3 License

The BSD 3-clause license allows you almost unlimited freedom with the software so long as you include the BSD copyright and license notice in it (found in Fulltext).

CHAPTER

THREE

INSTALLATION

This is the recommended way to install uttrs.

3.1 Installing with pip

Make sure that the Python interpreter can load uttrs code. The most convenient way to do this is to use virtualenv, virtualenvwrapper, and pip.

After setting up and activating the virtualenv, run the following command:

```
$ pip install uttrs
...
```

That should be it all.

3.2 Installing the development version

If you'd like to be able to update your uttrs code occasionally with the latest bug fixes and improvements, follow these instructions:

Make sure that you have Git installed and that you can run its commands from a shell. (Enter *git help* at a shell prompt to test this.)

Check out uttrs main development branch like so:

```
$ git clone https://github.com/quatrope/uttrs
...
```

This will create a directory *uttrs* in your current directory.

Then you can proceed to install with the commands

```
$ cd uttrs
$ pip install -e .
...
```

CHAPTER

FOUR

DOCUMENTATION

The full documentation of the project are available in https://uttrs.readthedocs.io/

CHAPTER

FIVE

CONTACT

For bugs or question please contact

Juan B. Cabral: jbcabral@unc.edu.ar

CHAPTER

SIX

QUICK START

The following piece of code is an example prototype of a Class representing a Galaxy. The Galaxy contains:

- three arrays (`x`, `y`, `z`) with particle positions, measured in *kiloparsecs* (`u.kpc`).
- three arrays (`vx`, `vy`, `vz`) for the particle velocities, measured in *Km/s* (`u.kms/u.s`).
- an array (`m`) of particle masses, expressed in *solar masses* (`u.M_sun`).
- a free text for note taking in `notes`.

In every case we would like to access to position, velocity and mass of the particles, with and without units (as `np.ndarray`). Suggested units in the information of the attributes behave like this:

- If the user makes the class instance without unit specification then default assumed unit is used.
- If, otherwise, another unit is used as input, it is validated the feasibility of the conversion to default unit.

```
import uttr

import astropy.units as u

@uttr.s
class Galaxy:
    x = uttr.ib(unit=u.kpc)
    y = uttr.ib(unit=u.kpc)
    z = uttr.ib(unit=u.kpc)

    vx = uttr.ib(unit=u.km/u.s)
    vy = uttr.ib(unit=u.km/u.s)
    vz = uttr.ib(unit=u.km/u.s)

    m = uttr.ib(unit=u.M_sun)

    notes = uttr.ib(converter=str)
```

CHAPTER

SEVEN

CREATING A GALAXY

```
>>> import numpy as np
>>> import astropy.units as u

# Creating the particle arrays
>>> x = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> y = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> z = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> vx = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> vy = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> vz = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)
>>> m = np.random.randint(1000, 10_000, size=5) + np.random.rand(5)

>>> gal = Galaxy(
...     x = x * u.kpc,  # kpc is the suggested unit
...     y = y * u.mpc,  # milliparsec is equivalent to kpc
...     z = z,  # we assume is the suggested kpc unit
...     vx = vx * (u.km/u.s), # the suggested unit
...     vy = vy * (u.km/u.s), # the suggested unit
...     vz = vz, # the suggested unit
...     m = m * u.M_sun, # the suggested unit
...     notes="a random galaxy made with random numbers")

>>> gal
Galaxy(
    x=<Quantity [5632.35740606, 1363.36235923, 3037.46794044, 2785.45299727, 2515.
↪35793673] kpc>,
    y=<Quantity [4457.3573917 , 2873.54575512, 7979.68745148, 5930.55394614, 5903.
↪63598164] mpc>,
    z=<Quantity [6122.35929872, 3740.22821927, 6859.42245056, 7119.8256744 , 3632.
↪74980958] kpc>,
    vx=<Quantity [7141.40469733, 5713.29552487, 5000.535142  , 9366.36402447, 2967.
↪2546077 ] km / s>,
    vy=<Quantity [8514.83018331, 1362.13309457, 1136.30959053, 1985.49551226, 3286.
↪69029298] km / s>,
    vz=<Quantity [6218.56279077, 2015.04638043, 9919.99579782, 1278.94359767, 7228.
↪21626876] km / s>,
    m=<Quantity [5640.62516958, 4070.66620947, 6106.583697  , 4063.39917315, 3028.
↪85393523] solMass>,
    notes='a random galaxy made with random numbers')
```

(continues on next page)

(continued from previous page)

```
# we can access al the attributes in the traditional python way
>>> gal.x
<Quantity [5632.35740606, 1363.36235923, 3037.46794044, 2785.45299727, 2515.35793673]␣
↪kpc>

>>> gal.vz  # z is now a km/s
<Quantity [6218.56279077, 2015.04638043, 9919.99579782, 1278.94359767, 7228.21626876] km␣
↪/ s>

# We stored y as mpc (milliparsec)
>>> gal.y
<Quantity [8093.44916403, 2198.55398718, 5464.79397835, 1860.72260272, 3636.64010118]␣
↪mpc>
```

CHAPTER

EIGHT

SIMPLE INTERACTION WITH NUMPY.NDARRAY

We can access all the same attributes declared with `uttr.ib` but coerced to the default unit as numpy array.

```
>>> gal.arr_.y
array([0.00809345, 0.00219855, 0.00546479, 0.00186072, 0.00363664])
```

The above code is equivalent to

```
>>> gal.y.to_value(u.kpc)
array([0.00809345, 0.00219855, 0.00546479, 0.00186072, 0.00363664])
```

CHAPTER

NINE

EQUIVALENT UNITS ERRORS

If we change the unit to something not equivalent to the default unit declares in `uttr.ib` an exception is raised.

Lets fot example define `x` as a kilogram (`u.kg`)

```
>>> gal = Galaxy(
...      x = x * u.kg,  # kg is not equivalent to kpc
...      y = y,
...      z = z,
...      vx = vx,
...      vy = vy,
...      vz = vz,
...      m = m,
...      notes="a random galaxy made with random numbers")

ValueError: Unit of attribute 'x' must be equivalent to 'kpc'.Found 'kg'.
```

CHAPTER

TEN

REFERENCES

10.1 Astropy

Price-Whelan, Adrian M., et al. "The Astropy project: Building an open-science project and status of the v2. 0 core package." The Astronomical Journal 156.3 (2018): 123.

CHAPTER

ELEVEN

CONTENTS:

11.1 Tutorial

This tutorial is intended to serve as a guide on how to create classes using *uttrs*

11.1.1 Interactive Version

Launch Binder for an interactive version of this tutorial!

11.1.2 Imports

Let's first import all necessary libraries at the top. You will generally need just three:

- `attr` (*attrs*) is the library uttrs is based on. Attrs creates classes with less boilerplate code.
- `astropy.units` is a library that contains all the machinery to deal with astronomical and physical units.
- `uttr` (*uttrs*) is the library we will explore on this tutorial.

  Note: This tutorial assumes knowledge on the aforementioned libraries. Please refer to the reference links at the end of this notebook for more information.

[1]:
```
import attr  # to use the .validators module
import uttr
import astropy.units as u
```

The Galaxy Class

We will create a stripped-down version of the Galaxy class from the Galaxy-Chop project.

It will have only 8 attributes. The first 7 will have units attached and will be implemented with `uttr.ib`. These are:

- `x`, `y`, `z`: The postions of the particles (typically stars) from the center of the galaxy measured in KiloParsecs (kpc).
- `vx`, `vy`, `vz`: The relative velocity components of the particles measured in km/s.
- `m`: Masses of the particles in units of solar masses ($M_\odot$).

The last attribute `notes` is a description text about the galaxy and can be implemented with the standar *attrs* library.

[2]:
```
@uttr.s
class Galaxy:
    x = uttr.ib(unit=u.kpc)
    y = uttr.ib(unit=u.kpc)
    z = uttr.ib(unit=u.kpc)

    vx = uttr.ib(unit=u.km / u.s)
    vy = uttr.ib(unit=u.km / u.s)
    vz = uttr.ib(unit=u.km / u.s)

    m = uttr.ib(unit=u.M_sun)

    notes = attr.ib(validator=attr.validators.instance_of(str))
```

Galaxy with Default Units

Now that we created our class, we can go ahead and create an object of type *Galaxy*.

To keep it simple, let's assume only 4 particles with totally arbitrary numbers on each attribute.

Part of *uttrs* power is its ability to assign default units when not provided, or to validate that the input unit is physically compatible with the given default.

Let's see first an example in which all units are assigned automatically.

[3]:
```
gal = Galaxy(
    x=[1, 1, 3, 4],
    y=[10, 2, 3, 100],
    z=[1, 1, 1, 1],
    vx=[1000, 1023, 2346, 1334],
    vy=[9956, 833, 954, 1024],
    vz=[1253, 956, 1054, 3568],
    m=[200, 100, 20, 5],
    notes="A random galaxy with arbitrary numbers.",
)
```

Let's verify that all attributes of the class were given the correct units.

[4]:
```
gal.x
```

[4]: $[1,\ 1,\ 3,\ 4]\ \mathrm{kpc}$

[5]:
```
gal.y
```

[5]: $[10,\ 2,\ 3,\ 100]\ \mathrm{kpc}$

[6]:
```
gal.vx
```

[6]: $[1000,\ 1023,\ 2346,\ 1334]\ \frac{\mathrm{km}}{\mathrm{s}}$

[7]:
```
gal.m
```

[7]: $[200,\ 100,\ 20,\ 5]\ \mathrm{M}_\odot$

[8]:
```
gal.notes
```

[8]:
```
'A random galaxy with arbitrary numbers.'
```

Galaxy with Explicit Units

A different alternative is to provide units compatible with the default unit. In this case, we have to be mindful of the phyisical equivalence of units with the ones given at the time the class was created.

For example, we could suggest that the dimension `z` be given in parsecs, `vy` in km/h and masses in kg.

[9]:
```
gal = Galaxy(
    x=[1, 1, 3, 4],
    y=[10, 2, 3, 100],
    z=[1000, 1000, 1000, 1000] * u.parsec,
    vx=[1000, 1023, 2346, 1334],
    vy=[9956, 833, 954, 1024] * (u.km / u.h),
    vz=[1253, 956, 1054, 3568],
    m=[200, 100, 20, 5] * u.kg,
    notes="A random galaxy with arbitrary numbers.",
)
```

As we note above, this works as expected without error. We can further access any of the attributes and verify that they keep the suggested units.

[10]:
```
gal.z  # parsecs
```

[10]: $[1000,\ 1000,\ 1000,\ 1000]\ \mathrm{pc}$

[11]:
```
gal.m  # kg
```

[11]: $[200,\ 100,\ 20,\ 5]\ \mathrm{kg}$

[12]:
```
gal.vx  # default km/s
```

[12]: $[1000,\ 1023,\ 2346,\ 1334]\ \frac{\mathrm{km}}{\mathrm{s}}$

[13]:
```
gal.vy # km/h
```

[13]: $[9956,\ 833,\ 954,\ 1024]\ \frac{\mathrm{km}}{\mathrm{h}}$

On the other hand, if we try to input a unit that is incompatible with the suggested input unit, a `ValueError` exception is raised.

To show this, let's try to assign `x` values with units of grams (g).

[14]:
```
gal = Galaxy(
    x=[1, 1, 3, 4] * u.g,
    y=[10, 2, 3, 100],
    z=[1000, 1000, 1000, 1000] * u.parsec,
    vx=[1000, 1023, 2346, 1334],
    vy=[9956, 833, 954, 1024] * (u.km / u.h),
    vz=[1253, 956, 1054, 3568],
    m=[200, 100, 20, 5] * u.kg,
```

(continues on next page)

(continued from previous page)

```
    notes="A random galaxy with arbitrary numbers.",
)
```

```
---------------------------------------------------------------------------
UnitConversionError                       Traceback (most recent call last)
~/proyectos/uttrs/src/uttr.py in validate_is_equivalent_unit(self, instance, attribute,␣
↪value)
    131         try:
--> 132             unity.to(self.unit)
    133         except u.UnitConversionError:

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/quantity.py in to(self, unit,
↪ equivalencies)
    688         unit = Unit(unit)
--> 689         return self._new_view(self._to_value(unit, equivalencies), unit)
    690

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/quantity.py in _to_
↪value(self, unit, equivalencies)
    659             equivalencies = self._equivalencies
--> 660         return self.unit.to(unit, self.view(np.ndarray),
    661                             equivalencies=equivalencies)

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/core.py in to(self, other,␣
↪value, equivalencies)
    986         else:
--> 987             return self._get_converter(other, equivalencies=equivalencies)(value)
    988

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/core.py in _get_
↪converter(self, other, equivalencies)
    917
--> 918             raise exc
    919

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/core.py in _get_
↪converter(self, other, equivalencies)
    902         try:
--> 903             return self._apply_equivalencies(
    904                 self, other, self._normalize_equivalencies(equivalencies))

~/proyectos/uttrs/lib/python3.8/site-packages/astropy/units/core.py in _apply_
↪equivalencies(self, unit, other, equivalencies)
    885
--> 886         raise UnitConversionError(
    887             "{} and {} are not convertible".format(

UnitConversionError: 'g' (mass) and 'kpc' (length) are not convertible

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-14-ab318fb696d1> in <module>
----> 1 gal = Galaxy(
      2     x=[1, 1, 3, 4] * u.g,
      3     y=[10, 2, 3, 100],
      4     z=[1000, 1000, 1000, 1000] * u.parsec,
      5     vx=[1000, 1023, 2346, 1334],

<attrs generated init __main__.Galaxy> in __init__(self, x, y, z, vx, vy, vz, m, notes)
      9     self.notes = notes
     10     if _config._run_validators is True:
---> 11         __attr_validator_x(self, __attr_x, self.x)
     12         __attr_validator_y(self, __attr_y, self.y)
     13         __attr_validator_z(self, __attr_z, self.z)

~/proyectos/uttrs/lib/python3.8/site-packages/attr/_make.py in __call__(self, inst, attr,
↪ value)
   2721     def __call__(self, inst, attr, value):
   2722         for v in self._validators:
-> 2723             v(inst, attr, value)
   2724
   2725

~/proyectos/uttrs/src/uttr.py in validate_is_equivalent_unit(self, instance, attribute,␣
↪value)
    133         except u.UnitConversionError:
    134             unit, aname, ufound = self.unit, attribute.name, value.unit
--> 135             raise ValueError(
    136                 f"Unit of attribute '{aname}' must be equivalent to '{unit}'."
    137                 f" Found '{ufound}'."

ValueError: Unit of attribute 'x' must be equivalent to 'kpc'. Found 'g'.
```

11.1.3 Automatic Cohersion of Units: Array Accessor

One powerful feauture of *uttrs* is the ability to easily transform all units to plain `numpy.ndarray`, using the default units.

This is achieved using the `uttr.array_accessor()` function. This allows for uniform access of attributes defined by uttrs, in a data structure that has faster access time than its counterpart with units.

By default the `@uttr.s` automataclly add an array accessor to decorated class. You can disabled this functionallity using the decorator like `@uttr.s(aaccessor=None)`, or change the name of the property with `@uttr.s(aaccessor="other_name")`.

Expanding on the previous example:

[ ]:
```
@uttr.s
class Galaxy:
    x = uttr.ib(unit=u.kpc)
    y = uttr.ib(unit=u.kpc)
    z = uttr.ib(unit=u.kpc)
```

(continues on next page)

(continued from previous page)

```
    vx = uttr.ib(unit=u.km / u.s)
    vy = uttr.ib(unit=u.km / u.s)
    vz = uttr.ib(unit=u.km / u.s)

    m = uttr.ib(unit=u.M_sun)

    notes = attr.ib(validator=attr.validators.instance_of(str))
```

Let's instantiate the class again with some parameters with custom units.

[ ]:
```
gal = Galaxy(
    x=[1, 1, 3, 4],
    y=[10, 2, 3, 100],
    z=[1000, 1000, 1000, 1000] * u.parsec,
    vx=[1000, 1023, 2346, 1334],
    vy=[9956, 833, 954, 1024] * (u.km / u.h),
    vz=[1253, 956, 1054, 3568],
    m=[200, 100, 20, 5] * u.kg,
    notes="A random galaxy with arbitrary numbers.",
)
```

If we now access `z` through our `arr_` accessor, *uttrs* will convert the values in parsec units to kiloparsecs and return a uniform numpy array.

[ ]:
```
gal.arr_.z
```

While `z` keeps its original unit.

[ ]:
```
gal.arr_.z
```

The same applies to `vy` and `m`.

[ ]:
```
gal.arr_.m
```

[ ]:
```
gal.arr_.vy
```

If we try to access a private attribute not from `uttr.ib`, an `AttributeError` exception is raised.

[ ]:
```
gal.arr_.notes
```

11.1.4 Using the `array_accessor`

It is a known issue that Astropy units can slow down complex computations.

To avoid this, developers usually choose to uniformize units and convert the values to numpy arrays to operate on them faster; reverting back to values with units at the end of the calculation.

As a helper, `array_accesor` will perform the transformation in a transparent way to the user, avoiding the need to replicate information regarding units.

For example, if we wanted to program code that generates a new Galaxy object with a single particle that is the average mean of all the rest, we could do something like this:

[ ]:
```
@uttr.s
class Galaxy:
    x = uttr.ib(unit=u.kpc)
    y = uttr.ib(unit=u.kpc)
    z = uttr.ib(unit=u.kpc)

    vx = uttr.ib(unit=u.km / u.s)
    vy = uttr.ib(unit=u.km / u.s)
    vz = uttr.ib(unit=u.km / u.s)

    m = uttr.ib(unit=u.M_sun)

    notes = attr.ib(validator=attr.validators.instance_of(str))

    def mean(self):
        x = np.mean(self.arr_.x)
        y = np.mean(self.arr_.y)
        z = np.mean(self.arr_.z)

        vx = np.mean(self.arr_.vx)
        vy = np.mean(self.arr_.vy)
        vz = np.mean(self.arr_.vz)

        m = np.mean(self.arr_.m)

        return Galaxy(
            x=x, y=y, z=z, vx=vx, vy=vy, vz=vz, m=m, notes=self.notes
        )
```

We could now create a galaxy with 1 million random elements and calculate the "average" galaxy.

[ ]:
```
import numpy as np

# Fix random seed
random = np.random.default_rng(seed=42)

size = 1_000_000

gal = Galaxy(
    x=random.random(size=size),
    y=random.random(size=size),
    z=random.random(size=size) * u.parsec,
    vx=random.random(size=size),
    vy=random.random(size=size),
    vz=random.random(size=size) * (u.km / u.h),
    m=random.random(size=size) * u.kg,
    notes="A random galaxy with arbitrary numbers.",
)
```

[ ]:
```
gal.mean()
```

To complete the example, let's see how would a `mean` method look like without `array_accessor`.

[ ]:
```
@uttr.s(aaccessor=None)
class Galaxy:
    x = uttr.ib(unit=u.kpc)
    y = uttr.ib(unit=u.kpc)
    z = uttr.ib(unit=u.kpc)

    vx = uttr.ib(unit=u.km / u.s)
    vy = uttr.ib(unit=u.km / u.s)
    vz = uttr.ib(unit=u.km / u.s)

    m = uttr.ib(unit=u.M_sun)

    notes = attr.ib(validator=attr.validators.instance_of(str))

    def mean(self):
        x = np.mean(self.x.to_value(u.kpc))
        y = np.mean(self.y.to_value(u.kpc))
        z = np.mean(self.z.to_value(u.kpc))

        vx = np.mean(self.vx.to_value(u.km / u.s))
        vy = np.mean(self.vy.to_value(u.km / u.s))
        vz = np.mean(self.vz.to_value(u.km / u.s))

        m = np.mean(self.m.to_value(u.M_sun))

        return Galaxy(
            x=x, y=y, z=z, vx=vx, vy=vy, vz=vz, m=m, notes=self.notes
        )
```

[ ]:
```
import datetime as dt
dt.date.today().isoformat()
```

[ ]:
```

```

11.2 uttrs API

uttrs bridge between attrs and Astropy units[1].

uttrs seeks to interoperate Classes defined using attrs and Astropy units in a simple manner with two main functionalities:

- `uttr.ib` which generates attributes sensitive to units.
- `uttr.array_accessor` which allows access to attributes linked to units, and transform them into numpy arrays.
- `uttr.s` a class decorator to automatically add the *array_accessor*.

[1] Price-Whelan, Adrian M., et al. "The Astropy project: Building an open-science project and status of the v2. 0 core package." The Astronomical Journal 156.3 (2018): 123.

References

class uttr.**ArrayAccessor**(*instance*)

Bases: object

Automatic converter of the uttrs attributes in numpy.ndarray.

Instances of ArrayAccessor (arr_) access to the attributes (defined with uttrs) of the provided instance, and if they are of atropy.units.Quantity type it converts them into numpy.ndarray.

If you try to access a attribute no defined by uttrs, an AttributeErrror is raised.

Examples

```
>>> @attr.s()
... class Foo:
...     quantity = uttr.ib(unit=u.km)
...     array = attr.ib()
```

```
>>> foo = Foo(
...     quantity=u.Quantity([1, 2]),
...     array=np.array([1, 2]),
... )
```

```
>>> arr_ = ArrayAccessor(foo)
```

```
>>> arr_
ArrayAccessor(
    Foo(quantity=<Quantity [1., 2.]>, array=array([1, 2]))
```

```
>>> arr_.quantity
array([1., 2., 3.])
```

```
>>> arr_.array
AttributeError("No uttr.Attribute 'array'")
```

class uttr.**UnitConverterAndValidator**(*unit: astropy.units.core.UnitBase*)

Bases: object

Converter and validator of astropy.units for attrs.

Parameters **unit** (*astropy.units.UnitBase*) – The base units for attribute default unit assignation and validation of inputs.

convert_if_dimensionless(*value*)

Assign a unit to a dimensionless object.

If the object already has a dimension it returns it without change

Examples

```
>>> uc = UnitConverter(u.km)
```

```
>>> uc.convert_if_dimensionless(1)  # dimensionless then convert
'<Quantity 1. km>'
```

```
>>> # the same object is returned
>>> uc.convert_if_dimensionless(1 * u.kpc)
'<Quantity 1. kpc>'
```

is_dimensionless(*v*)
Return true if v is dimensionless.

to_array(*v*)
Convert the quantity to an array of the given unit.

unit: astropy.units.core.UnitBase

validate_is_equivalent_unit(*instance*, *attribute*, *value*)
Validate that the unit equivalence with. the configured unit.

This method follows the suggested signature by attrs validators.

- the instance that's being validated (aka self),
- the attribute that it's validating, and finally
- the value that is passed for it.

Raises ValueError: – If the value has a non-equivalent dimesion to unit.

uttr.**array_accessor**()
Provide an ArrayAccessor attribute to an attrs based class.

This new attribute allows access to any other uttrs defined attribute or of the class. It converts it to the default unit of the attribute and afterward to a *numpy.ndarray*.

If you try to access an attribute no defined by uttrs, an `AttributeErrror` is raised.

Example

```
>>> @attr.s()
... class Foo:
...     q = uttr.ib(unit=u.kg)
...     a = attr.ib()
...     arr_ = array_accessor()
```

```
>>> foo = Foo(q=[1, 2, 3] * u.kg, a=np.array([1, 2, 3]))
>>> foo
Foo(q=<Quantity [1., 2., 3.] kg>, a=array([1, 2, 3]))
```

```
>>> foo.q
<Quantity [1., 2., 3.] kg>
```

```
>>> foo.arr_.q
array([1., 2., 3.])
```

```
>>> foo.a
array([1, 2, 3])
```

```
>>> foo.arr_.a
array([1, 2, 3])
```

uttr.**attribute**(*unit: Optional[astropy.units.core.UnitBase] = None*, ***kwargs*)

Create a new attribute with converters and validators for a given unit.

Parameters

- **unit** (*u.UnitBase or None*) – The unit to use in the converters and the attribute validator. If it's None, the call is equivalent to `attr.ib(**kwargs)`
- **kwargs** – Extra parameter of attr.ib()

Example

```
>>> @attr.s()
... class Foo:
...     p = unit_attribute(unit=(u.km / u.s))
>>>> Foo(p=[1, 2, 3])
Foo(p=<Quantity [1., 2., 3.] km / s>)
```

@attr.s() >>> class Foo: … p = unit_attribute(unit=(u.km / u.h))

```
>>> Foo(p=[1, 2, 3])
Foo(p=<Quantity [1., 2., 3.] km / h>)
```

```
>>> @attr.s()
... class Foo:
...     p = unit_attribute(unit=(u.km / u.s))
```

```
>>> Foo(p=[1, 2, 3] * u.km / u.h)
Foo(p=<Quantity [1., 2., 3.] km / h>)
```

```
>>> Foo(p=[1, 2, 3] * u.kpc)
ValueError: Unit of attribute 'p' must be equivalent to 'km / s'.
Found 'kpc'.
```

uttr.**ib**(*unit: Optional[astropy.units.core.UnitBase] = None*, ***kwargs*)

Equivalent to `uttr.attribute` to use like *attr.ib*.

uttr.**s**(*maybe_cls=None*, ***, *aaccessor='arr_'*, ***kwargs*)

Class decorator to automatically add an array accessor to the class.

The behaviour is the same as attr.s function but also automatically creates an `uttrs.ArrayAccessor` property with defined by *aaccessor*.

Parameters

- **aaccessor** (*str or None, default 'arr_'*) – Name of the array accessor property. If is None, no property is added.
- **maybe_cls** (*class or None, default None.*) – Same behavior of `attr.s()` maybe_cls parameter.
- **kwargs** – Same parameter as `attr.s()`.

Examples

The next two codes are equivalent

```
import astropy.units as u
import uttr

@attr.s
class Foo:
    attribute = uttr.ib(unit=u.K)
    arr_ = uttr.array_accessor()
```

```
import astropy.units as u
import uttr

@uttr.s
class Foo:
    attribute = uttr.ib(unit=u.K)
```

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX