uttrs

Release 0.4.1

JBCabral and QuatroPe

Nov 22, 2020

CONTENTS

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

uttrs is mainly two 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.

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 attr
import uttr

import astropy.units as u

@attr.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))

    arr_ = uttr.array_accessor()
```

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')

# 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>
```

(continues on next page)

(continued from previous page)

```
>>> 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

Este tutorial busca servir de una guia para la creación de clases utilizando *uttrs*.

DRAFT!

11.1.1 Versión interactiva

Puede ejecutarse este mismo tutorial de manera interactiba en Binder.

11.1.2 Imports

En primer lugar es necesario importar todas las librerias que vamos a utilizar. En general, son solo tres:

- `attr` (*attrs*) que es la libreria en la cual se basa uttrs para crear clases con menos boiler plate.
- `astropy.units` La cual contiene todo el marco de utilidades para tratar con unidades físicas/astronómicas.
- `uttr` (*uttrs*) La librería que corresponde este tutorial.

  Nota: Este tutorial asume un conocimiento sobre estas librerías, una serie de enlaces de referencias pueden encontrarse al final de la página.

[1]:
```
import attr
import uttr

import astropy.units as u
```

The galaxy class

La clase que vamos a crear, consiste en una simplicación de la clase Galaxia del proyecot Galaxy-Chop.

Solo tiene 8 atributos y solo los primeros 7 tienen unidades y por lo tanto seran implementadas con la funcion `uttr.ib` de la librería. Estos son`x`, `y`, `z` son las posiciones de las particulas/estrellas en KiloParsecs (`kpc`) ; `vx`, `vy`, `vz` las velocidades correspondientes a las particulas (Km/s); `m` su masa en masas solares ($M_\odot$). Todos

Finalmente `notes` texto libre sobre las galaxias y pueden ser implementadas con la librería *attrs* estandar

[2]:
```
@attr.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

Finalmente con la clase ya disponible podemos proceder a crear un objeto del tipo *Galaxy*,

Por cuestiones de simplicidadad, vamos a asumir solo 4 particulas con numeros totalmente arbitrarios en cada atributo.

Parte de la utilidad de usar uttrs, es la capacidad que tiene la libreria para agregar unidades por defecto automáticamente, o validar que la unidad ingresada sea equivalente.

Empecemos con un objeto en el cual todas las unidades se asignan autoaticamente

[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 made with random numbers",
)
```

Si nos fijamos en cualquier atributo de la clase, vamos a ver que todas las unidades se agregaron de manera correcta

[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 made with random numbers'
```

Galaxy with explicit units

Otra alternativa consiste en proveer unidades compatibles con las establecidas, hay que tener en cuenta que estas unidades tienen que ser equivalentes a las propuestas en la creacion de la clase.

Por ejemplo podemos sugerir que la dimension `z` este dada en parsecs, `vy` en Km/h y las masas en 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 made with random numbers",
)
```

Como se nota en el ejemplo, esto funciona perfectamente, y ningun error aparece. Es mas podemos acceder a culquiera de los atributos presentes y todos mantienen las unidades sugeridas o explicitas

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

Por otro lado si por error cargamos un valor que tiene una unidad que no es equivalente a la sugerida, se lanza un error de valor (`ValueError`)

Para demostrar esto vamos a extender el ejemplo, tratando de asignar a `x` un valor expresado en gramos (`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,
    notes="a random galaxy made with random numbers",
)
```

```
---------------------------------------------------------------------------
UnitConversionError                       Traceback (most recent call last)
~/proyectos/uttrs/src/uttr.py in validate_is_equivalent_unit(self, instance,␣
↪attribute, value)
    135         try:
--> 136             unity.to(self.unit)
```

(continues on next page)

(continued from previous page)

```
    137         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)
<ipython-input-14-0d84924447d4> 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)
```

(continues on next page)

(continued from previous page)

```
     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)
    137         except u.UnitConversionError:
    138             unit, aname, ufound = self.unit, attribute.name, value.unit
--> 139             raise ValueError(
    140                 f"Unit of attribute '{aname}' must be equivalent to '{unit}'."
    141                 f" Found '{ufound}'."

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

11.1.3 Automatic cohersion of units: Array Accessor

El mayor poder de *uttrs* es la capacidad de transformar de manera sencilla todas las unidades a `numpy.ndarray` planos, utilizando las unidades por defecto.

Para esto se provee de una funcion `uttr.array_accessor()` la cual permite acceder a los atributos definidos por *uttrs* de manera uniforme en una estructura de datos mas veloz que las que possen unidad.

Para agregar esta característica, se debe agregar un atributo extra a la clase que se iguale a `uttr.array_accessor()`. Se propone utilizar el nombre `arr_`

Extendiendo el ejemplo anterior

[15]:
```
@attr.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))

    arr_ = uttr.array_accessor()  # el accessor
```

Ahora volvemos a instanciar la clase con algunos parámetros con unidades custom

[16]:
```
gal = Galaxy(
    x=[1, 1, 3, 4],
    y=[10, 2, 3, 100],
    z=[1000, 1000, 1000, 1000] * u.parsec,
    vx=[1000, 1023, 2346, 1334],
```

(continues on next page)

(continued from previous page)

```
    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 made with random numbers",
)
```

ahora si accedemos a `z` a travez de `arr_`, uttrs se encargara de convertir los parsecs en kiloparsecs y luego convertirlo en un numpy array

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

```
[17]: array([1., 1., 1., 1.])
```

Mientras que `z` mantiene sus unidades originales

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

```
[18]: array([1., 1., 1., 1.])
```

Lo mismo si accedemos a `vy` y `m`

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

```
[19]: array([1.00582884e-28, 5.02914422e-29, 1.00582884e-29, 2.51457211e-30])
```

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

```
[20]: array([2.76555556, 0.23138889, 0.265     , 0.28444444])
```

Tratar de acceder a un atributo privado o que no sea un `uttr.ib`, lanza un error del tipo `AttributeError`

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-92911293967b> in <module>
----> 1 gal.arr_.notes

~/proyectos/uttrs/src/uttr.py in __getattr__(self, a)
    278             return arr
    279
--> 280         raise AttributeError(f"No uttr.Attribute '{a}'")
    281
    282

AttributeError: No uttr.Attribute 'notes'
```

11.1.4 El uso de `array_accessor`

Es conocido que las unidades de Astropy son lentas cuando los calculos son complejos.

Para evitar esto, los desarrolladores optan por unificar las unidades y luego convertir los valores a arrays de numpy para operarlos mas rapidamente; y al final se vuelve a asignas las unidades.

Para evitar esto, `array_accesor` realiza toda esta trasnformacion transparentemente para el usuario, evitando la necesidad de replicar informacion sobre las unidades.

Por ejemplo, si quisieramos programar un codigo que genere un nuevo objeto galaxia con una sola particula, promedio de las demas, el código seria el siguiente:

[22]:
```
@attr.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))

    arr_ = uttr.array_accessor()  # el accessor

    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
        )
```

Ahora podemos crear una galaxia de 1 millon de elementos aleatorios y calcular la galaxia “media”

[23]:
```
import numpy as np

# fijamos la semilla random
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 made with random numbers",
)
```

[24]:
```
gal.mean()
```

[24]:
```
Galaxy(x=<Quantity 0.50002648 kpc>, y=<Quantity 0.49981983 kpc>, z=<Quantity 0.
↪00049982 kpc>, vx=<Quantity 0.49976787 km / s>, vy=<Quantity 0.50029902 km / s>, vz=
↪<Quantity 0.00013897 km / s>, m=<Quantity 2.5120227e-31 solMass>, notes='a random␣
↪galaxy made with random numbers')
```

Por completitud se ejemplifica a continuacion como seria el miemos codigo de `mean` pero sin usar `array_accessor`

[25]:
```
@attr.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))

    arr_ = uttr.array_accessor()  # el accessor

    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
        )
```

[26]:
```
import datetime as dt
dt.datetime.now()
```

[26]:
```
datetime.datetime(2020, 11, 21, 20, 52, 24, 347474)
```

[ ]:
```

```

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.

[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: astropy.units.core.UnitBase*, ***kwargs*)

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

Parameters

- **unit** (*u.UnitBase*) – The unit to use in the converters and the attribute validator
- **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: astropy.units.core.UnitBase*, ***kwargs*)

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

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

u

INDEX