pint Documentation

*Release 0.10.dev0*

Hernan E. Grecco

Nov 01, 2019
Contents

1 Quick Installation 3
2 Design principles 5
3 User Guide 7
4 More information 41
5 One last thing 97
Python Module Index 99
Index 101
Pint is a Python package to define, operate and manipulate **physical quantities**: the product of a numerical value and a unit of measurement. It allows arithmetic operations between them and conversions from and to different units.

It is distributed with a comprehensive list of physical units, prefixes and constants. Due to its modular design, you can extend (or even rewrite!) the complete list without changing the source code. It supports a lot of numpy mathematical operations **without monkey patching or wrapping numpy**.

It has a complete test coverage. It runs in Python 2.7 and 3.3+ with no other dependency. It is licensed under BSD.

It is extremely easy and natural to use:

```python
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> 3 * ureg.meter + 4 * ureg.cm
<Quantity(3.04, 'meter')>
```

and you can make good use of numpy if you want:

```python
>>> import numpy as np
>>> [3, 4] * ureg.meter + [4, 3] * ureg.cm
<Quantity([ 3.04 4.03], 'meter')>
>>> np.sum(_)
<Quantity(7.07, 'meter')>
```
Quick Installation

To install Pint, simply:

```bash
$ pip install pint
```

or utilizing conda, with the conda-forge channel:

```bash
$ conda install -c conda-forge pint
```

and then simply enjoy it!
Although there are already a few very good Python packages to handle physical quantities, no one was really fitting my needs. Like most developers, I programmed Pint to scratch my own itches.

**Unit parsing:** prefixed and pluralized forms of units are recognized without explicitly defining them. In other words: as the prefix kilo and the unit meter are defined, Pint understands kilometers. This results in a much shorter and maintainable unit definition list as compared to other packages.

**Standalone unit definitions:** units definitions are loaded from a text file which is simple and easy to edit. Adding and changing units and their definitions does not involve changing the code.

**Advanced string formatting:** a quantity can be formatted into string using PEP 3101 syntax. Extended conversion flags are given to provide symbolic, LaTeX and pretty formatting. Unit name translation is available if Babel is installed.

**Free to choose the numerical type:** You can use any numerical type (fraction, float, decimal, numpy.ndarray, etc). NumPy is not required but supported.

**NumPy integration:** When you choose to use a NumPy ndarray, its methods and ufuncs are supported including automatic conversion of units. For example numpy.arccos(q) will require a dimensionless q and the units of the output quantity will be radian.

**Uncertainties integration:** transparently handles calculations with quantities with uncertainties (like 3.14±0.01) meter via the uncertainties package.

**Handle temperature:** conversion between units with different reference points, like positions on a map or absolute temperature scales.

**Small codebase:** easy to maintain codebase with a flat hierarchy.

**Dependency free:** it depends only on Python and its standard library.

**Python 2 and 3:** a single codebase that runs unchanged in Python 2.7+ and Python 3.3+

**Pandas integration:** Thanks to Pandas Extension Types it is now possible to use Pint with Pandas. Operations on DataFrames and between columns are units aware, providing even more convenience for users of Pandas DataFrames. For full details, see the [pint-pandas Jupyter notebook](#).
When you choose to use a NumPy ndarray, its methods and ufuncs are supported including automatic conversion of units. For example `numpy.arccos(q)` will require a dimensionless $q$ and the units of the output quantity will be radian.
3.1 Installation

Pint has no dependencies except Python itself. It runs on Python 2.7 and 3.3+. You can install it (or upgrade to the latest version) using pip:

```
$ pip install -U pint
```

That’s all! You can check that Pint is correctly installed by starting up python, and importing pint:

```
>>> import pint
>>> pint.__version__
```

**Note:** If you have an old system installation of Python and you don’t want to mess with it, you can try Anaconda CE. It is a free Python distribution by Continuum Analytics that includes many scientific packages. To install pint from the conda-forge channel instead of through pip use:

```
$ conda install -c conda-forge pint
```

You can check the installation with the following command:

```
>>> pint.test()    # doctest: +SKIP
```

On Arch Linux, you can alternatively install Pint from the Arch User Repository (AUR). The latest release is available as python-pint, and packages tracking the master branch of the GitHub repository are available as python-pint-git and python2-pint-git.

3.1.1 Getting the code

You can also get the code from PyPI or GitHub. You can either clone the public repository:
3.2 Tutorial

3.2.1 Converting Quantities

Pint has the concept of Unit Registry, an object within which units are defined and handled. You start by creating your registry:

```python
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
```

If no parameter is given to the constructor, the unit registry is populated with the default list of units and prefixes. You can now simply use the registry in the following way:

```python
>>> distance = 24.0 * ureg.meter
>>> print(distance)
24.0 meter
>>> time = 8.0 * ureg.second
>>> print(time)
8.0 second
>>> print(repr(time))
<Quantity(8.0, 'second')>
```

In this code `distance` and `time` are physical quantity objects (``Quantity``). Physical quantities can be queried for their magnitude, units, and dimensionality:

```python
>>> print(distance.magnitude)
24.0
>>> print(distance.units)
meter
>>> print(distance.dimensionality)
[length]
```

and can handle mathematical operations between:

```python
>>> speed = distance / time
>>> print(speed)
3.0 meter / second
```

As unit registry knows about the relationship between different units, you can convert quantities to the unit of choice:
This method returns a new object leaving the original intact as can be seen by:

```python
>>> print(speed)
3.0 meter / second
```

If you want to convert in-place (i.e. without creating another object), you can use the `ito` method:

```python
>>> speed.ito(ureg.inch / ureg.minute)
>>> speed
<Quantity(7086.614173228345, 'inch / minute')>

>>> print(speed)
7086.614173228345 inch / minute
```

If you ask Pint to perform an invalid conversion:

```python
>>> speed.to(ureg.joule)
Traceback (most recent call last):
...  
DimensionalityError: Cannot convert from 'inch / minute' ([length] / [time]) to 'joule' ([length] ** 2 * [mass] / [time] ** 2)
```

Sometimes, the magnitude of the quantity will be very large or very small. The method `to_compact` can adjust the units to make the quantity more human-readable.

```python
>>> wavelength = 1550 * ureg.nm
>>> frequency = (ureg.speed_of_light / wavelength).to('Hz')

>>> print(frequency)
193414489032258.03 hertz

>>> print(frequency.to_compact())
193.41448903225802 terahertz
```

There are also methods `to_base_units` and `ito_base_units` which automatically convert to the reference units with the correct dimensionality:

```python
>>> height = 5.0 * ureg.foot + 9.0 * ureg.inch

>>> print(height)
5.75 foot

>>> print(height.to_base_units())
1.7526 meter

>>> print(height)
5.75 foot

>>> height.ito_base_units()

>>> print(height)
1.7526 meter
```

There are also methods `to_reduced_units` and `ito_reduced_units` which perform a simplified dimensional reduction, combining units with the same dimensionality but otherwise keeping your unit definitions intact.

```python
>>> density = 1.4 * ureg.gram / ureg.cm**3

>>> volume = 10*ureg.cc

>>> mass = density * volume

>>> print(mass)
14.0 cc * gram / centimeter ** 3

>>> print(mass.to_reduced_units())
(continues on next page)
If you want pint to automatically perform dimensional reduction when producing new quantities, the UnitRegistry accepts a parameter `auto_reduce_dimensions`. Dimensional reduction can be slow, so auto-reducing is disabled by default.

In some cases it is useful to define physical quantities objects using the class constructor:

```python
>>> Q_ = ureg.Quantity
>>> Q_(1.78, ureg.meter) == 1.78 * ureg.meter
True
```

(I tend to abbreviate Quantity as `Q_`.) The built-in parser recognizes prefixed and pluralized units even though they are not in the definition list:

```python
>>> distance = 42 * ureg.kilometers
>>> print(distance)
42 kilometer
>>> print(distance.to(ureg.meter))
42000.0 meter
```

If you try to use a unit which is not in the registry:

```python
>>> speed = 23 * ureg.snail_speed
Traceback (most recent call last):
... UndefinedUnitError: 'snail_speed' is not defined in the unit registry
```

You can add your own units to the registry or build your own list. More info on that *Defining units*

### 3.2.2 String parsing

Pint can also handle units provided as strings:

```python
>>> 2.54 * ureg.parse_expression('centimeter')
<Quantity(2.54, 'centimeter')>
```

or using the registry as a callable for a short form for `parse_expression`

```python
>>> 2.54 * ureg('centimeter')
<Quantity(2.54, 'centimeter')>
```

or using the `Quantity` constructor:

```python
>>> Q_(2.54, 'centimeter')
<Quantity(2.54, 'centimeter')>
```

Numbers are also parsed, so you can use an expression:

```python
>>> ureg('2.54 * centimeter')
<Quantity(2.54, 'centimeter')>
```
or:

```python
>>> Q_('2.54 * centimeter')
<Quantity(2.54, 'centimeter')>
```

or leave out the * altogether:

```python
>>> Q_('2.54cm')
<Quantity(2.54, 'centimeter')>
```

This enables you to build a simple unit converter in 3 lines:

```python
>>> user_input = '2.54 * centimeter to inch'
>>> src, dst = user_input.split(' to ')
>>> Q_(src).to(dst)
<Quantity(1.0, 'inch')>
```

Dimensionless quantities can also be parsed into an appropriate object:

```python
>>> ureg('2.54')
2.54
>>> type(ureg('2.54'))
<class 'float'>
```

or

```python
>>> Q_('2.54')
<Quantity(2.54, 'dimensionless')>
>>> type(Q_('2.54'))
<class 'pint.quantity.build_quantity_class.<locals>.Quantity'>
```

**Note:** Pint’s rule for parsing strings with a mixture of numbers and units is that **units are treated with the same precedence as numbers.**

For example, the unit of

```python
>>> Q_('3 l / 100 km')
<Quantity(0.03, 'kilometer * liter')>
```

may be unexpected first but is a consequence of applying this rule. Use brackets to get the expected result:

```python
>>> Q_('3 l / (100 km)')
<Quantity(0.03, 'liter / kilometer')>
```

**Note:** Since version 0.7, Pint **does not** use `eval` under the hood. This change removes the serious security problems that the system is exposed to when parsing information from untrusted sources.

### 3.2.3 String formatting

Pint’s physical quantities can be easily printed:
>>> accel = 1.3 * ureg['meter/second**2']
>>> # The standard string formatting code
>>> print('The str is {!s}'.format(accel))
The str is 1.3 meter / second ** 2
>>> # The standard representation formatting code
>>> print('The repr is {!r}'.format(accel))
The repr is <Quantity(1.3, 'meter / second ** 2')>
>>> # Accessing useful attributes
>>> print('The magnitude is {0.magnitude} with units {0.units}'.format(accel))
The magnitude is 1.3 with units meter / second ** 2

Pint supports float formatting for numpy arrays as well:

```python
>>> accel = np.array([-1.1, 1e-6, 1.2505, 1.3]) * ureg['meter/second**2']
>>> # float formatting numpy arrays
>>> print('The array is {:.2f}'.format(accel))
The array is [-1.10 0.00 1.25 1.30] meter / second ** 2
>>> # scientific form formatting with unit pretty printing
>>> print('The array is {:+.2E~P}'.format(accel))
The array is [-1.10E+00 +1.00E-06 +1.25E+00 +1.30E+00] m/s^2
```

Pint also supports ‘f-strings’ from python>=3.6:

```python
>>> accel = 1.3 * ureg['meter/second**2']
>>> print(f'The str is {accel}')
The str is 1.3 meter / second ** 2
>>> print(f'The str is {accel:.3e}')
The str is 1.300e+00 meter / second ** 2
>>> print(f'The str is {accel:~}')
The str is 1.3 m / s ** 2
>>> print(f'The str is {accel:~.3e}')
The str is 1.300e+00 m / s ** 2
>>> print(f'The str is {accel:~H}')
The str is 1.3 m/s^2
```

But Pint also extends the standard formatting capabilities for unicode and LaTeX representations:

```python
>>> accel = 1.3 * ureg['meter/second**2']
>>> # Pretty print
>>> 'The pretty representation is {:P}'.format(accel)
'The pretty representation is 1.3 meter/second^2'
>>> # Latex print
>>> 'The latex representation is {:L}'.format(accel)
'The latex representation is \(1.3\ \frac{\text{meter}}{\text{second}^{2}}\)'
>>> # HTML print
>>> 'The HTML representation is {:H}'.format(accel)
'The HTML representation is 1.3 meter/second<sup>2</sup>
```

**Note:** In Python 2, run from __future__ import unicode_literals or prefix pretty formatted strings with `u` to prevent UnicodeEncodeError.

If you want to use abbreviated unit names, prefix the specification with `~`:

```python
>>> 'The str is {~}'.format(accel)
'The str is 1.3 m / s ** 2'
```
The pretty representation is 1.3 m/s²

The same is true for LaTeX (L) and HTML (H) specs.

Note: The abbreviated unit is drawn from the unit registry where the 3rd item in the equivalence chain (i.e., 1 = 2 = 3) will be returned when the prefix ‘~’ is used. The 1st item in the chain is the canonical name of the unit.

The formatting specs (i.e., ‘L’, ‘H’, ‘P’) can be used with Python string ‘formatting syntax’ for custom float representations. For example, scientific notation:

.. doctest::

    >>> 'Scientific notation: {:.3e~L}'.format(accel)
    'Scientific notation: 1.300\times 10^{0}\ \frac{\mathrm{m}}{\mathrm{s}^{2}}'

Pint also supports the LaTeX siunitx package:

>>> accel = 1.3 * ureg['meter/second**2']
>>> # siunitx Latex print
>>> print('The siunitx representation is {:Lx}'.format(accel))
The siunitx representation is \SI{1.3}{\meter\per\second\squared}

Additionally, you can specify a default format specification:

>>> 'The acceleration is {}' .format(accel)
'The acceleration is 1.3 meter / second ** 2'
>>> ureg.default_format = 'P'
>>> 'The acceleration is {}' .format(accel)
'The acceleration is 1.3 meter/second²'

Finally, if Babel is installed you can translate unit names to any language

>>> accel.format_babel(locale='fr_FR')
'1.3 mètre par seconde²'

### 3.2.4 Using Pint in your projects

If you use Pint in multiple modules within your Python package, you normally want to avoid creating multiple instances of the unit registry. The best way to do this is by instantiating the registry in a single place. For example, you can add the following code to your package `__init__.py`:

```python
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity
```

Then in `yourmodule.py` the code would be:

```python
from . import ureg, Q_
length = 10 * ureg.meter
my_speed = Q_(20, 'm/s')
```
If you are pickling and unpickling Quantities within your project, you should also define the registry as the application registry:

```
from pint import UnitRegistry, set_application_registry
ureg = UnitRegistry()
set_application_registry(ureg)
```

**Warning:** There are no global units in Pint. All units belong to a registry and you can have multiple registries instantiated at the same time. However, you are not supposed to operate between quantities that belong to different registries. Never do things like this:

```python
>>> q1 = 10 * UnitRegistry().meter
>>> q2 = 10 * UnitRegistry().meter
>>> q1 + q2
Traceback (most recent call last):
... 
ValueError: Cannot operate with Quantity and Quantity of different registries.
>>> id(q1._REGISTRY) == id(q2._REGISTRY)
False
```

### 3.3 NumPy support

The magnitude of a Pint quantity can be of any numerical type and you are free to choose it according to your needs. In numerical applications, it is quite convenient to use NumPy ndarray and therefore they are supported by Pint.

First, we import the relevant packages:

```python
>>> import numpy as np
>>> from pint import UnitRegistry
"```

and then we create a quantity the standard way

```python
>>> legs1 = Q_(np.asarray([3., 4.]), 'meter')
>>> print(legs1)
[ 3. 4.] meter
```

or we use the property that Pint converts iterables into NumPy ndarrays to simply write:

```python
>>> legs1 = [3., 4.] * ureg.meter
>>> print(legs1)
[ 3. 4.] meter
```

All usual Pint methods can be used with this quantity. For example:

```python
>>> print(legs1.to('kilometer'))
[ 0.003 0.004] kilometer
>>> print(legs1.dimensionality)
[length]
>>> print(legs1.to('joule'))
Traceback (most recent call last):
```

(continues on next page)
DimensionalityError: Cannot convert from 'meter' ([length]) to 'joule' ([length] ** 2 * [mass] / [time] ** 2)

NumPy functions are supported by Pint. For example if we define:

```python
>>> legs2 = [400., 300.] * ureg.centimeter
>>> print(legs2)
[ 400. 300.] centimeter
```

we can calculate the hypotenuse of the right triangles with legs1 and legs2.

```python
>>> hyps = np.hypot(legs1, legs2)
>>> print(hyps)
[ 5. 5.] meter
```

Notice that before the `np.hypot` was used, the numerical value of legs2 was internally converted to the units of legs1 as expected.

Similarly, when you apply a function that expects angles in radians, a conversion is applied before the requested calculation:

```python
>>> angles = np.arccos(legs2/hyps)
>>> print(angles)
[ 0.64350111 0.92729522] radian
```

You can convert the result to degrees using the corresponding NumPy function:

```python
>>> print(np.rad2deg(angles))
[ 36.86989765 53.13010235] degree
```

Applying a function that expects angles to a quantity with a different dimensionality results in an error:

```python
>>> np.arccos(legs2)
Traceback (most recent call last):
...  
  DimensionalityError: Cannot convert from 'centimeter' ([length]) to 'dimensionless'
```

### 3.3.1 Support

The following `ufuncs` can be applied to a Quantity object:

- **Math operations**: add, subtract, multiply, divide, logaddexp, logaddexp2, true_divide, floor_divide, negative, remainder mod, fmod, absolute, rint, sign, conj, exp, exp2, log, log2, log10, expm1, log1p, sqrt, square, reciprocal
- **Trigonometric functions**: sin, cos, tan, arcsin, arccos, arctan, arctan2, hypot, sinh, cosh, tanh, arcsinh, arccosh, arctanh, deg2rad, rad2deg
- **Comparison functions**: greater, greater_equal, less, less_equal, not_equal, equal
- **Floating functions**: isreal, iscomplex, isfinite, isnan, signbit, copysign, nextafter, modf, ldexp, frexp, fmod, floor, ceil, trunc

And the following `NumPy ndarray methods` and functions:

#### 3.3. NumPy support
• sum, fill, reshape, transpose, flatten, ravel, squeeze, take, put, repeat, sort, argsort, diagonal, compress, nonzero, 
  searchsorted, max, argmax, min, argmin, ptp, clip, round, trace, cumsum, mean, var, std, prod, cumprod, conj, 
  conjugate, flatten

*Quantity* is not a subclass of *ndarray*. This might change in the future, but for this reason functions that call 
numpy.asanyarray are currently not supported. These functions are:

• unwrap, trapz, diff, ediff1d, fix, gradient, cross, ones_like

### 3.3.2 Comments

What follows is a short discussion about how NumPy support is implemented in Pint’s *Quantity* Object.

For the supported functions, Pint expects certain units and attempts to convert the input (or inputs). For example, the 
argument of the exponential function (*numpy.exp*) must be dimensionless. Units will be simplified (converting the 
magnitude appropriately) and *numpy.exp* will be applied to the resulting magnitude. If the input is not dimensionless, 
a *DimensionalityError* exception will be raised.

In some functions that take 2 or more arguments (e.g. *arctan2*), the second argument is converted to the units of the 
first. Again, a *DimensionalityError* exception will be raised if this is not possible.

This behaviour introduces some performance penalties and increased memory usage. Quantities that must be converted 
to other units require additional memory and CPU cycles. On top of this, all *ufuncs* are implemented in the *Quantity* 
class by overriding *__array_wrap__*, a NumPy hook that is executed after the calculation and before returning the 
value. To our knowledge, there is no way to signal back to NumPy that our code will take care of the calculation. 
For this reason the calculation is actually done twice: first in the original *ndarray* and then in the one that has been 
converted to the right units. Therefore, for numerically intensive code, you might want to convert the objects first and 
then use directly the magnitude.

### 3.4 Temperature conversion

Unlike meters and seconds, the temperature units fahrenheits and celsius are non-multiplicative units. These tempera-
ture units are expressed in a system with a reference point, and relations between temperature units include not only a 
scaling factor but also an offset. Pint supports these type of units and conversions between them. The default definition 
file includes fahrenheits, celsius, kelvin and rankine abbreviated as degF, degC, degK, and degR.

For example, to convert from celsius to fahrenheit:

```python
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
>>> Q_ = ureg.Quantity
>>> home = Q_(25.4, ureg.degC)
>>> print(home.to('degF'))
77.7200004 degF
```

or to other kelvin or rankine:

```python
>>> print(home.to('kelvin'))
298.55 kelvin
>>> print(home.to('degR'))
537.39 degR
```

Additionally, for every non-multiplicative temperature unit in the registry, there is also a *delta* counterpart to specify 
differences. Absolute units have no *delta* counterpart. For example, the change in celsius is equal to the change in 
kelvin, but not in fahrenheit (as the scaling factor is different).
Subtraction of two temperatures given in offset units yields a \textit{delta} unit:

\begin{verbatim}
>>> Q_(25.4, ureg.degC) - Q_(10., ureg.degC)
<Quantity(15.4, 'delta_degC')>
\end{verbatim}

You can add or subtract a quantity with \textit{delta} unit and a quantity with offset unit:

\begin{verbatim}
>>> Q_(25.4, ureg.degC) + Q_(10., ureg.delta_degC)
<Quantity(35.4, 'degC')>
>>> Q_(25.4, ureg.degC) - Q_(10., ureg.delta_degC)
<Quantity(15.4, 'degC')>
\end{verbatim}

If you want to add a quantity with absolute unit to one with offset unit, like here

\begin{verbatim}
>>> heating_rate = 0.5 * ureg.kelvin/ureg.min
>>> Q_(10., ureg.degC) + heating_rate * Q_(30, ureg.min)
Traceback (most recent call last):
  ... 
  OffsetUnitCalculusError: Ambiguous operation with offset unit (degC, kelvin).
\end{verbatim}

you have to avoid the ambiguity by either converting the offset unit to the absolute unit before addition

\begin{verbatim}
>>> Q_(10., ureg.degC).to(ureg.kelvin) + heating_rate * Q_(30, ureg.min)
<Quantity(298.15, 'kelvin')>
\end{verbatim}

or convert the absolute unit to a \textit{delta} unit:

\begin{verbatim}
>>> Q_(10., ureg.degC) + heating_rate.to('delta_degC/min') * Q_(30, ureg.min)
<Quantity(25.0, 'degC')>
\end{verbatim}

In contrast to subtraction, the addition of quantities with offset units is ambiguous, e.g. for 10 \textit{degC} + 100 \textit{degC} two different result are reasonable depending on the context, 110 \textit{degC} or 383.15 \textdegree{}C (= 283.15 K + 373.15 K). Because of this ambiguity pint raises an error for the addition of two quantities with offset units (since pint-0.6).

Quantities with \textit{delta} units are multiplicative:

\begin{verbatim}
>>> speed = 60. * ureg.delta_degC / ureg.min
>>> print(speed.to('delta_degC/second'))
1.0 delta_degC / second
\end{verbatim}

However, multiplication, division and exponentiation of quantities with offset units is problematic just like addition. Pint (since version 0.6) will by default raise an error when a quantity with offset unit is used in these operations. Due to this quantities with offset units cannot be created like other quantities by multiplication of magnitude and unit but have to be explicitly created:

\begin{verbatim}
>>> ureg = UnitRegistry()
>>> home = 25.4 * ureg.degC
Traceback (most recent call last):
  ... 
  OffsetUnitCalculusError: Ambiguous operation with offset unit (degC).
>>> Q_(25.4, ureg.degC)
<Quantity(25.4, 'degC')>
\end{verbatim}
As an alternative to raising an error, pint can be configured to work more relaxed via setting the UnitRegistry parameter `autoconvert_offset_to_baseunit` to true. In this mode, pint behaves differently:

- Multiplication of a quantity with a single offset unit with order +1 by a number or ndarray yields the quantity in the given unit.

```python
>>> ureg = UnitRegistry(autoconvert_offset_to_baseunit = True)
>>> T = 25.4 * ureg.degC
>>> T
<Quantity(25.4, 'degC')>
```

- Before all other multiplications, all divisions and in case of exponentiation\(^1\) involving quantities with offset-units, pint will convert the quantities with offset units automatically to the corresponding base unit before performing the operation.

```python
>>> 1/T
<Quantity(0.00334952269302, '1 / kelvin')>
>>> T * 10 * ureg.meter
<Quantity(527.15, 'kelvin * meter')>
```

You can change the behaviour at any time:

```python
>>> ureg.autoconvert_offset_to_baseunit = False
>>> 1/T
Traceback (most recent call last):
... OffsetUnitCalculusError: Ambiguous operation with offset unit (degC).
```

The parser knows about *delta* units and uses them when a temperature unit is found in a multiplicative context. For example, here:

```python
>>> print(ureg.parse_units('degC/meter'))
delta_degC / meter
```

but not here:

```python
>>> print(ureg.parse_units('degC'))
degC
```

You can override this behaviour:

```python
>>> print(ureg.parse_units('degC/meter', as_delta=False))
degC / meter
```

Note that the magnitude is left unchanged:

```python
>>> Q_(10, 'degC/meter')
<Quantity(10, 'delta_degC / meter')>
```

To define a new temperature, you need to specify the offset. For example, this is the definition of the celsius and fahrenheit:

```python
degC = degK; offset: 273.15 = celsius
degF = 5 / 9 * degK; offset: 255.372222 = fahrenheit
```

You do not need to define *delta* units, as they are defined automatically.

\(^1\) If the exponent is +1, the quantity will not be converted to base unit but remains unchanged.
3.5 Wrapping and checking functions

In some cases you might want to use pint with a pre-existing web service or library which is not units aware. Or you might want to write a fast implementation of a numerical algorithm that requires the input values in some specific units.

For example, consider a function to return the period of the pendulum within a hypothetical physics library. The library does not use units, but instead requires you to provide numerical values in certain units:

```python
>>> from simple_physics import pendulum_period
>>> help(pendulum_period)
Help on function pendulum_period in module simple_physics:

pendulum_period(length)
    Return the pendulum period in seconds. The length of the pendulum
    must be provided in meters.

>>> pendulum_period(1)
2.0064092925890407
```

This behaviour is very error prone, in particular when combining multiple libraries. You could wrap this function to use Quantities instead:

```python
>>> from pint import UnitRegistry

ureg = UnitRegistry()

>>> def mypp_caveman(length):
...     return pendulum_period(length.to(ureg.meter).magnitude) * ureg.second

and:

>>> mypp_caveman(100 * ureg.centimeter)
<Quantity(2.0064092925890407, 'second')>
```

Pint provides a more convenient way to do this:

```python
>>> mypp = ureg.wraps(ureg.second, ureg.meter)(pendulum_period)
```

Or in the decorator format:

```python
>>> @ureg.wraps(ureg.second, ureg.meter)
... def mypp(length):
...     return pendulum_period(length)

>>> mypp(100 * ureg.centimeter)
<Quantity(2.0064092925890407, 'second')>
```

`wraps` takes 3 input arguments:

- **ret**: the return units. Use None to skip conversion.
- **args**: the inputs units for each argument, as an iterable. Use None to skip conversion of any given element.
- **strict**: if **True** all convertible arguments must be a Quantity and others will raise a ValueError (True by default)

### 3.5.1 Strict Mode

By default, the function is wrapped in *strict* mode. In this mode, the input arguments assigned to units must be a Quantities.
To enable using non-Quantity numerical values, set strict to False.

In this mode, the value is assumed to have the correct units.

### 3.5.2 Multiple arguments or return values

For a function with more arguments, use a tuple:

```
>>> from simple_physics import pendulum_period2
>>> help(pendulum_period2)
Help on function pendulum_period2 in module simple_physics:

pendulum_period2(length, swing_amplitude)
Return the pendulum period in seconds. The length of the pendulum
must be provided in meters. The swing_amplitude must be in radians.
```

```
>>> mypp2 = ureg.wraps(ureg.second, (ureg.meter, ureg.radians))(pendulum_period2)
```

Or if the function has multiple outputs:

```
>>> mypp3 = ureg.wraps((ureg.second, ureg.meter / ureg.second),
... (ureg.meter, ureg.radians))(pendulum_period_maxspeed)
```

If there are more return values than specified units, None is assumed for the extra outputs. For example, given the NREL SOLPOS calculator that outputs solar zenith, azimuth and air mass, the following wrapper assumes no units for airmass:

```
@ureg.wraps(('deg', 'deg'), ('deg', 'deg', 'millibar', 'degC'))
def solar_position(lat, lon, press, tamb, timestamp):
    return zenith, azimuth, airmass
```

### 3.5.3 Optional arguments

For a function with named keywords with optional values, use a tuple for all arguments:

```
>>> @ureg.wraps(ureg.second, (ureg.meters, ureg.meters/ureg.second**2))
... def calculate_time_to_fall(height, gravity=Q_(9.8, 'm/s^2'), verbose=False):
...     """Calculate time to fall from a height h."
```
... By default, the gravity is assumed to be earth gravity, but it can be modified.
... 
... \( d = \frac{1}{2} g \times t^2 \) 
... \( t = \sqrt{\frac{2 \times d}{g}} \) 
... 
... if verbose: print(str(t) + " seconds to fall") 
... return t 

```python
>>> lunar_module_height = Q_(22, 'feet') + Q_(11, 'inches')
>>> calculate_time_to_fall(lunar_module_height, verbose=True)
1.1939473204801092 seconds to fall
<Quantity(1.1939473204801092, 'second')>
>>> moon_gravity = Q_(1.625, 'm/s^2')
>>> tcalculate_time_to_fall(lunar_module_height, moon_gravity)
<Quantity(2.932051001760214, 'second')>
```

### 3.5.4 Specifying relations between arguments

In certain cases, you may not be concerned with the actual units and only care about the unit relations among arguments.

This is done using a string starting with the equal sign `=`:

```python
>>> @ureg.wraps('=A**2', ('=A', '=A'))
... def sqsum(x, y):
...     return x * x + 2 * x * y + y * y
```

which can be read as the first argument \( x \) has certain units (we labeled them \( A \)), the second argument \( y \) has the same units as the first \( A \) again). The return value has the unit of \( x \) squared \( (A**2) \)

You can use more than one label:

```python
>>> @ureg.wraps('=A*B', ('=A', '=B'))
... def get_displacement(time, rate=Q_(1, 'm/s')):
...     return time * rate
```

With optional arguments

```python
>>> @ureg.wraps('=A*B', ('=A', '=B'))
... def get_displacement(time, rate=Q_(1, 'm/s')):
...     return time * rate
...
>>> get_displacement(Q_(2, 's'))
<Quantity(2, 'meter')>
>>> get_displacement(Q_(2, 's'), Q_(1, 'deg/s'))
<Quantity(2, 'degree')>
```

### 3.5.5 Ignoring an argument or return value

To avoid the conversion of an argument or return value, use None

3.5. Wrapping and checking functions
3.6 Checking dimensionality

When you want pint quantities to be used as inputs to your functions, pint provides a wrapper to ensure units are of correct type - or more precisely, they match the expected dimensionality of the physical quantity.

Similar to wraps(), you can pass None to skip checking of some parameters, but the return parameter type is not checked.

```python
>>> mypp3 = ureg.wraps((ureg.second, None), ureg.meter)(pendulum_period_error)
```

In the decorator format:

```python
>>> @ureg.check('[length]')
... def pendulum_period(length):
...     return 2*math.pi*math.sqrt(length/G)
```

If you just want to check the dimensionality of a quantity, you can do so with the built-in ‘check’ function.

```python
>>> distance = 1 * ureg.m
>>> distance.check('[length]')
True
>>> distance.check('[time]')
False
```

3.7 Plotting with Matplotlib

Matplotlib is a Python plotting library that produces a wide range of plot types with publication-quality images and support for typesetting mathematical formulas. Starting with Matplotlib 2.0, Quantity instances can be used with matplotlib’s support for units when plotting. To do so, the support must be manually enabled on a UnitRegistry:

```python
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> ureg.setup_matplotlib()
```

This support can also be disabled with:

```python
>>> ureg.setup_matplotlib(False)
```

This allows plotting quantities with different units:

```python
import matplotlib.pyplot as plt
import numpy as np
import pint
ureg = pint.UnitRegistry()
ureg.setup_matplotlib(True)
y = np.linspace(0, 30) * ureg.miles
x = np.linspace(0, 5) * ureg.hours
```
This also allows controlling the actual plotting units for the x and y axes:

```python
import matplotlib.pyplot as plt
import numpy as np
import pint

ereg = pint.UnitRegistry()
ereg.setup_matplotlib(True)

y = np.linspace(0, 30) * ureg.miles
x = np.linspace(0, 5) * ureg.hours

fig, ax = plt.subplots()
ax.yaxis.set_units(ureg.inches)
ax.xaxis.set_units(ureg.seconds)
ax.plot(x, y, 'tab:blue')
ax.axhline(26400 * ureg.feet, color='tab:red')
ax.axvline(120 * ureg.minutes, color='tab:green')
```

For more information, visit the Matplotlib home page.

### 3.8 Serialization

In order to dump a `Quantity` to disk, store it in a database or transmit it over the wire you need to be able to serialize and then deserialize the object.

The easiest way to do this is by converting the quantity to a string:

```python
>>> import pint
>>> ureg = pint.UnitRegistry()

>>> duration = 24.2 * ureg.years
>>> duration
<Quantity(24.2, 'year')>

>>> serialized = str(duration)

>>> print(serialized)
24.2 year
```

Remember that you can easily control the number of digits in the representation as shown in *String formatting*.

You dump/store/transmit the content of serialized (`'24.2 year'`). When you want to recover it in another process/machine, you just:

```python
>>> import pint

>>> ureg = pint.UnitRegistry()

>>> duration = ureg('24.2 year')

>>> print(duration)
24.2 year
```

Notice that the serialized quantity is likely to be parsed in another registry as shown in this example. Pint Quantities do not exist on their own but they are always related to a `UnitRegistry`. Everything will work as expected if both
registries, are compatible (e.g. they were created using the same definition file). However, things could go wrong if the registries are incompatible. For example, `year` could not be defined in the target registry. Or what is even worse, it could be defined in a different way. Always have to keep in mind that the interpretation and conversion of Quantities are UnitRegistry dependent.

In certain cases, you want a binary representation of the data. Python’s standard algorithm for serialization is called **Pickle**. Pint quantities implement the magic `__reduce__` method and therefore can be **Pickled** and **Unpickled**. However, you have to bear in mind, that the **application registry** is used for unpickling and this might be different from the one that was used during pickling.

By default, the application registry is one initialized with `defaults_en.txt`; in other words, the same as what you get when creating a `pint.UnitRegistry` without arguments and without adding any definitions afterwards.

If your application is fine just using `defaults_en.txt`, you don’t need to worry further.

If your application needs a single, global registry with custom definitions, you must make sure that it is registered using `pint.set_application_registry()` before unpickling anything. You may use `pint.get_application_registry()` to get the current instance of the application registry.

Finally, if you need multiple custom registries, it’s impossible to correctly unpickle `pint.Quantity` or `pint.Unit` objects. The best way is to create a tuple with the magnitude and the units:

```python
>>> to_serialize = duration.to_tuple()
>>> print(to_serialize)
(24.2, (('year', 1.0),))
```

And then you can just pickle that:

```python
>>> import pickle
>>> serialized = pickle.dumps(to_serialize, -1)
```

To unpickle, just

```python
>>> loaded = pickle.loads(serialized)
>>> ureg.Quantity.from_tuple(loaded)
<Quantity(24.2, 'year')>
```

(To pickle to and from a file just use the dump and load method as described in _Pickle)_

You can use the same mechanism with any serialization protocol, not only with binary ones. (In fact, version 0 of the Pickle protocol is ASCII). Other common serialization protocols/packages are `json`, `yaml`, `shelve`, `hdf5` (or via `PyTables`) and `dill`. Notice that not all of these packages will serialize properly the magnitude (which can be any numerical type such as `numpy.ndarray`).

Using the `serialize` package you can load and read from multiple formats:

```python
>>> from serialize import dump, load, register_class
>>> register_class(ureg.Quantity, ureg.Quantity.to_tuple, ureg.Quantity.from_tuple)
>>> dump(duration, 'output.yaml')
>>> r = load('output.yaml')
```

(You can use the `serialize` docs for more information)

### 3.9 Buckingham Pi Theorem

Buckingham π theorem states that an equation involving \( n \) number of physical variables which are expressible in terms of \( k \) independent fundamental physical quantities can be expressed in terms of \( p = n - k \) dimensionless parameters.
To start with a very simple case, consider that you want to find a dimensionless quantity involving the magnitudes \(V\), \(T\) and \(L\) with dimensions \([\text{length}]/[\text{time}]\), \([\text{time}]\) and \([\text{length}]\) respectively.

```python
>>> from pint import pi_theorem
>>> pi_theorem({'V': '[length]/[time]', 'T': '[time]', 'L': '[length]'}

[{'V': 1.0, 'T': 1.0, 'L': -1.0}]
```

The result indicates that a dimensionless quantity can be obtained by multiplying \(V\) by \(T\) and the inverse of \(L\).

Which can be pretty printed using the Pint formatter:

```python
>>> from pint import formatter
>>> result = pi_theorem({'V': '[length]/[time]', 'T': '[time]', 'L': '[length]'})
>>> print(formatter(result[0].items()))
T * V / L
```

You can also apply the Buckingham \(\pi\) theorem associated to a Registry. In this case, you can use derived dimensions such as speed:

```python
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
>>> ureg.pi_theorem({'V': '[speed]', 'T': '[time]', 'L': '[length]'}

[{'V': 1.0, 'T': 1.0, 'L': -1.0}]
```

or unit names:

```python
>>> ureg.pi_theorem({'V': 'meter/second', 'T': 'second', 'L': 'meter'})

[{'V': 1.0, 'T': 1.0, 'L': -1.0}]
```

or quantities:

```python
>>> Q_ = ureg.Quantity
>>> ureg.pi_theorem({'V': Q_(1, 'meter/second'), ... 'T': Q_(1, 'second'), ... 'L': Q_(1, 'meter')})

[{'V': 1.0, 'T': 1.0, 'L': -1.0}]
```

3.9.1 Application to the pendulum

There are 3 fundamental physical units in this equation: time, mass, and length, and 4 dimensional variables, \(T\) (oscillation period), \(M\) (mass), \(L\) (the length of the string), and \(g\) (earth gravity). Thus we need only 4 - 3 = 1 dimensionless parameter.

```python
>>> ureg.pi_theorem({'T': '[time]', ... 'M': '[mass]', ... 'L': '[length]', ... 'g': '[acceleration]'})

[{'T': 2.0, 'g': 1.0, 'L': -1.0}]
```

which means that the dimensionless quantity is:

\[
\Pi = \frac{gT^2}{L}
\]

and therefore:

\[
T = constant \sqrt{\frac{L}{g}}
\]
(In case you wonder, the constant is equal to $2 \pi$, but this is outside the scope of this help)

### 3.9.2 Pressure loss in a pipe

What is the pressure loss $p$ in a pipe with length $L$ and diameter $D$ for a fluid with density $d$, and viscosity $m$ travelling with speed $v$? As pressure, mass, volume, viscosity and speed are defined as derived dimensions in the registry, we only need to explicitly write the density dimensions.

```python
>>> ureg.pi_theorem({'p': '[pressure]',
      ... 'L': '[length]',
      ... 'D': '[length]',
      ... 'd': '[mass]/[volume]',
      ... 'm': '[viscosity]',
      ... 'v': '[speed]'
      ... })
[['p': 1.0, 'm': -2.0, 'd': 1.0, 'L': 2.0}, {'v': 1.0, 'm': -1.0, 'd': 1.0, 'L': 1.0},
 → {'L': -1.0, 'D': 1.0}]
```

The second dimensionless quantity is the Reynolds Number

### 3.10 Contexts

If you work frequently on certain topics, you will probably find the need to convert between dimensions based on some pre-established (physical) relationships. For example, in spectroscopy you need to transform from wavelength to frequency. These are incompatible units and therefore Pint will raise an error if you do this directly:

```python
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> q = 500 * ureg.nm
>>> q.to('Hz')
Traceback (most recent call last):
... DimensionalityError: Cannot convert from 'nanometer' ([length]) to 'hertz' (1 / [time])
```

You probably want to use the relation \( frequency = speed\_of\_light / wavelength \):

```python
>>> (ureg.speed_of_light / q).to('Hz')
<Quantity(5.99584916e+14, 'hertz')>
```

To make this task easy, Pint has the concept of contexts which provides conversion rules between dimensions. For example, the relation between wavelength and frequency is defined in the spectroscopy context (abbreviated \( sp \)). You can tell pint to use this context when you convert a quantity to different units.

```python
>>> q.to('Hz', 'spectroscopy')
<Quantity(5.99584916e+14, 'hertz')>
```

or with the abbreviated form:

```python
>>> q.to('Hz', 'sp')
<Quantity(5.99584916e+14, 'hertz')>
```

Contexts can be also enabled for blocks of code using the with statement:
If you need a particular context in all your code, you can enable it for all operations with the registry:

```python
>>> ureg.enable_contexts('sp')
```

To disable the context, just call:

```python
>>> ureg.disable_contexts()
```

### 3.10.1 Enabling multiple contexts

You can enable multiple contexts:

```python
>>> q.to('Hz', 'sp', 'boltzmann')
<Quantity(5.99584916e+14, 'hertz')>
```

This works also using the `with` statement:

```python
>>> with ureg.context('sp', 'boltzmann):
...     q.to('Hz')
<Quantity(5.99584916e+14, 'hertz')>
```

or in the registry:

```python
>>> ureg.enable_contexts('sp', 'boltzmann')
>>> q.to('Hz')
<Quantity(5.99584916e+14, 'hertz')>
```

If a conversion rule between two dimensions appears in more than one context, the one in the last context has precedence. This is easy to remember if you think that the previous syntax is equivalent to nest contexts:

```python
>>> with ureg.context('sp'):
...     with ureg.context('boltzmann'):
...         q.to('Hz')
<Quantity(5.99584916e+14, 'hertz')>
```

### 3.10.2 Parameterized contexts

Contexts can also take named parameters. For example, in the spectroscopy you can specify the index of refraction of the medium \(n\). In this way you can calculate, for example, the wavelength in water of a laser which on air is 530 nm.

```python
>>> wl = 530. * ureg.nm
>>> f = wl.to('Hz', 'sp')
>>> f.to('nm', 'sp', n=1.33)
<Quantity(398.496240602, 'nanometer')>
```

Contexts can also accept Pint Quantity objects as parameters. For example, the ‘chemistry’ context accepts the molecular weight of a substance (as a Quantity with dimensions of [mass]/[substance]) to allow conversion between moles and mass.
3.10.3 Ensuring context when calling a function

Pint provides a decorator to make sure that a function called is done within a given context. Just like before, you have to provide as argument the name (or alias) of the context and the parameters that you wish to set.

```
>>> w1 = 530. * ureg.nm
>>> @ureg.with_context('sp', n=1.33)
... def f(wl):
...     return wl.to('Hz').magnitude
>>> f(wl)
398.496240602
```

This decorator can be combined with `wraps` or `check` decorators described in *Wrapping and checking functions*.

3.10.4 Defining contexts in a file

Like all units and dimensions in Pint, *contexts* are defined using an easy to read text syntax. For example, the definition of the spectroscopy context is:

```
@context(n=1) spectroscopy = sp
# n index of refraction of the medium.
[length] <-> [frequency]: speed_of_light / n / value
[frequency] -> [energy]: planck_constant * value
[energy] -> [frequency]: value / planck_constant
@end
```

The `@context` directive indicates the beginning of the transformations which are finished by the `@end` statement. You can optionally specify parameters for the context in parenthesis. All parameters are named and default values are mandatory. Multiple parameters are separated by commas (like in a python function definition). Finally, you provide the name of the context (e.g. spectroscopy) and, optionally, a short version of the name (e.g. sp) separated by an equal sign. See the definition of the ‘chemistry’ context in default_en.txt for an example of a multiple-parameter context.

Conversions rules are specified by providing source and destination dimensions separated using a colon (:) from the equation. A special variable named `value` will be replaced by the source quantity. Other names will be looked first in the context arguments and then in registry.

A single forward arrow (->) indicates that the equations is used to transform from the first dimension to the second one. A double arrow (<>->) is used to indicate that the transformation operates both ways.

Context definitions are stored and imported exactly like custom units definition file (and can be included in the same file as unit definitions). See “Defining units” for details.

3.10.5 Defining contexts programmatically

You can create `Context` object, and populate the conversion rules using python functions. For example:

```
>>> ureg = pint.UnitRegistry()
>>> c = pint.Context('ab')
>>> c.add_transformation('[length]', '[time]',
```

... lambda ureg, x: x / ureg.speed_of_light)
>>> c.add_transformation('[time]', '[length]',
... lambda ureg, x: x * ureg.speed_of_light)

ureg.add_context(c)

ureg("1 s").to("km", "ab")
299792.458 kilometer

It is also possible to create anonymous contexts without invoking add_context:

>>> c = pint.Context()
...
>>> ureg("1 s").to("km", c)
299792.458 kilometer

3.11 Using Measurements

Measurements are the combination of two quantities: the mean value and the error (or uncertainty). The easiest ways to generate a measurement object is from a quantity using the plus_minus operator.

>>> import numpy as np
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
>>> print(book_length)
(20.0 +/- 2.0) centimeter

You can inspect the mean value, the absolute error and the relative error:

>>> print(book_length.value)
20.0 centimeter
>>> print(book_length.error)
2.0 centimeter
>>> print(book_length.rel)
0.1

You can also create a Measurement object giving the relative error:

>>> book_length = (20. * ureg.centimeter).plus_minus(.1, relative=True)
>>> print(book_length)
(20.0 +/- 2.0) centimeter

Measurements support the same formatting codes as Quantity. For example, to pretty print a measurement with 2 decimal positions:

>>> print('{:.02fP}'.format(book_length))
(20.00 ± 2.00) centimeter

Mathematical operations with Measurements, return new measurements following the Propagation of uncertainty rules.

>>> print(2 * book_length)
(40.0 +/- 4.0) centimeter
>>> width = (10 * ureg.centimeter).plus_minus(1)
>>> print('{:.02f}'.format(book_length + width))
(30.00 +/- 2.24) centimeter

Note: only linear combinations are currently supported.

## 3.12 Defining units

### 3.12.1 In a definition file

To define units in a persistent way you need to create a unit definition file. Such files are simple text files in which the units are defined as function of other units. For example this is how the minute and the hour are defined in `default_en.txt`:

```
hour = 60 * minute = h = hr
minute = 60 * second = min
```

It is quite straightforward, isn’t it? We are saying that `minute` is 60 `seconds` and is also known as `min`.

1. The first word is always the canonical name.
2. Next comes the definition (based on other units).
3. Next, optionally, there is the unit symbol.
4. Finally, again optionally, a list of aliases, separated by equal signs. If one wants to specify aliases but not a symbol, the symbol should be conventionally set to `_`; e.g.:

```
millennium = 1e3 * year = _ = millennia
```

The order in which units are defined does not matter, Pint will resolve the dependencies to define them in the right order. What is important is that if you transverse all definitions, a reference unit is reached. A reference unit is not defined as a function of another units but of a dimension. For the time in `default_en.txt`, this is the `second`:

```
second = [time] = s = sec
```

By defining `second` as equal to a string `time` in square brackets we indicate that:

- `time` is a physical dimension.
- `second` is a reference unit.

The ability to define basic physical dimensions as well as reference units allows to construct arbitrary units systems.

Pint is shipped with a default definition file named `default_en.txt` where `en` stands for English. You can add your own definitions to the end of this file but you will have to be careful to merge when you update Pint. An easier way is to create a new file (e.g. `mydef.txt`) with your definitions:

```
dog_year = 52 * day = dy
```

and then in Python, you can load it as:

```python
>>> from pint import UnitRegistry
>>> # First we create the registry.
>>> ureg = UnitRegistry()
```
If you make a translation of the default units or define a completely new set, you don’t want to append the translated definitions so you just give the filename to the constructor:

```python
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry('/your/path/to/default_es.txt')
```

In the definition file, prefixes are identified by a trailing dash:

```
yocto- = 10.0**-24 = y-
```

It is important to note that prefixed defined in this way can be used with any unit, including non-metric ones (e.g. kiloinch is valid for Pint). This simplifies definitions files enormously without introducing major problems. Pint, like Python, believes that we are all consenting adults.

Derived dimensions are defined as follows:

```
[density] = [mass] / [volume]
```

Note that primary dimensions don’t need to be declared; they can be defined for the first time as part of a unit definition. Finally, one may add aliases to an already existing unit definition:

```
@alias
meter = metro = metr
```

This is particularly useful when one wants to enrich definitions from defaults_en.txt with new aliases from a custom file. It can also be used for translations (like in the example above) as long as one is happy to have the localized units automatically converted to English when they are parsed.

### 3.12.2 Programatically

You can easily add units, dimensions, or aliases to the registry programmatically. Let’s add a dog_year (sometimes written as dy) equivalent to 52 (human) days:

```python
>>> from pint import UnitRegistry
>>> # We first instantiate the registry.
>>> # If we do not provide any parameter, the default unit definitions are used.
>>> ureg = UnitRegistry()
>>> Q_ = ureg.Quantity

# Here we add the unit
>>> ureg.define('dog_year = 52 * day = dy')

# We create a quantity based on that unit and we convert to years.
>>> lassie_lifespan = Q_(10, 'year')
>>> print(lassie_lifespan.to('dog_years'))
70.23888438100961 dog_year
```

Note that we have used the name `dog_years` even though we have not defined the plural form as an alias. Pint takes care of that, so you don’t have to. Plural forms that aren’t simply built by adding a ’s’ suffix to the singular form should be explicitly stated as aliases (see for example `millennia` above).

You can also add prefixes programmatically:

### 3.12. Defining units
where the number indicates the multiplication factor.

Same for aliases and derived dimensions:

```python
>>> ureg.define('@alias meter = metro = metr')
>>> ureg.define('[hypervolume] = [length ** 4]')
```

**Warning**: Units, prefixes, aliases and dimensions added programmatically are forgotten when the program ends.

### 3.13 Optimizing Performance

Pint can impose a significant performance overhead on computationally-intensive problems. The following are some suggestions for getting the best performance.

**Note**: Examples below are based on the IPython shell (which provides the handy `%timeit` extension), so they will not work in a standard Python interpreter.

#### 3.13.1 Use magnitudes when possible

It’s significantly faster to perform mathematical operations on magnitudes (even though you’re still using pint to retrieve them from a quantity object).

```python
In [1]: from pint import UnitRegistry
In [2]: ureg = UnitRegistry()
In [3]: q1 = ureg('1m')
In [5]: q2 = ureg('2m')
In [6]: %timeit (q1-q2)
100000 loops, best of 3: 7.9 µs per loop
In [7]: %timeit (q1.magnitude-q2.magnitude)
1000000 loops, best of 3: 356 ns per loop
```

This is especially important when using pint Quantities in conjunction with an iterative solver, such as the `brentq` method from scipy:

```python
In [1]: from scipy.optimize import brentq
In [2]: def foobar_with_quantity(x):
   # find the value of x that equals q2
   # assign x the same units as q2
   qx = ureg(str(x)+str(q2.units))
   # compare the two quantities, then take their magnitude because
```
# brentq requires a dimensionless return type
return (qx - q2).magnitude

In [3]: def foobar_with_magnitude(x):
    # find the value of x that equals q2
    # don't bother converting x to a quantity, just compare it with q2's magnitude
    return x - q2.magnitude

In [4]: %timeit brentq(foobar_with_quantity, 0, q2.magnitude)
1000 loops, best of 3: 310 µs per loop

In [5]: %timeit brentq(foobar_with_magnitude, 0, q2.magnitude)
1000000 loops, best of 3: 1.63 µs per loop

Bear in mind that altering computations like this loses the benefits of automatic unit conversion, so use with care.

### 3.13.2 A safer method: wrapping

A better way to use magnitudes is to use pint’s wraps decorator (See Wrapping and checking functions). By decorating a function with wraps, you pass only the magnitude of an argument to the function body according to units you specify. As such this method is safer in that you are sure the magnitude is supplied in the correct units.

In [1]: import pint
In [2]: ureg = pint.UnitRegistry()
In [3]: import numpy as np
In [4]: def f(x, y):
    return (x - y) / (x + y) * np.log(x/y)
In [5]: @ureg.wraps(None, ('meter', 'meter'))
def g(x, y):
    return (x - y) / (x + y) * np.log(x/y)
In [6]: a = 1 * ureg.meter
In [7]: b = 1 * ureg.centimeter
In [8]: %timeit f(a, b)
1000 loops, best of 3: 312 µs per loop
In [9]: %timeit g(a, b)
1000000 loops, best of 3: 65.4 µs per loop

### 3.14 Different Unit Systems (and default units)

Pint Unit Registry has the concept of system, which is a group of units

```python
>>> import pint
>>> ureg = pint.UnitRegistry(system='mks')
```
This has an effect in the base units. For example:

```python
>>> q = 3600. * ureg.meter / ureg.hour
>>> q.to_base_units()
<Quantity(1.0, 'meter / second')>
```

But if you change to cgs:

```python
>>> ureg.default_system = 'cgs'
>>> q.to_base_units()
<Quantity(100.0, 'centimeter / second')>
```
or more drastically to:

```python
>>> ureg.default_system = 'imperial'
>>> '{:.3f}'.format(q.to_base_units())
'1.094 yard / second'
```

**Warning:** In versions previous to 0.7 to_base_units returns quantities in the units of the definition files (which are called root units). For the definition file bundled with pint this is meter/gram/second. To get back this behaviour use to_root_units.set ureg.system = None

You can also use system to narrow down the list of compatible units:

```python
>>> ureg.default_system = 'mks'
>>> ureg.get_compatible_units('meter')
frozenset({<Unit('light_year')>, <Unit('angstrom')>})
```
or for imperial units:

```python
>>> ureg.default_system = 'imperial'
>>> ureg.get_compatible_units('meter')
frozenset({<Unit('thou')>, <Unit('league')>, <Unit('nautical_mile')>, <Unit('inch')>, ...
  <Unit('mile')>, <Unit('yard')>, <Unit('foot')>})
```

You can check which unit systems are available:

```python
>>> dir(ureg.sys)
['US', 'cgs', 'imperial', 'mks']
```

Or which units are available within a particular system:

```python
>>> dir(ureg.sys.imperial)
['UK_hundredweight', 'UK_ton', 'acre_foot', 'cubic_foot', 'cubic_inch', 'cubic_yard', ...
  'drachm', 'foot', 'grain', 'imperial_barrel', 'imperial_bushel', 'imperial_cup', ...
  'imperial_fluid_drachm', 'imperial_fluid_ounce', 'imperial_gallon', 'imperial_gill', ...
  'imperial_peck', 'imperial_pint', 'imperial_quart', 'inch', 'long_hunderweight', ...
  'long_ton', 'mile', 'ounce', 'pound', 'quarter', 'short_hunderdweight', 'short_ton', ...
  'square_foot', 'square_inch', 'square_mile', 'square_yard', 'stone', 'yard']
```

Notice that this give you the opportunity to choose within units with colliding names:
3.15 Pandas support

It is convenient to use the Pandas package when dealing with numerical data, so Pint provides PintArray. A PintArray is a Pandas Extension Array, which allows Pandas to recognise the Quantity and store it in Pandas DataFrames and Series.

3.15.1 Basic example

This example will show the simplest way to use pandas with pint and the underlying objects. It’s slightly fiddly as you are not reading from a file. A more normal use case is given in Reading a csv.

First some imports

```python
[1]: import pandas as pd
  import pint
```

Next, we create a DataFrame with PintArrays as columns.

```python
[2]: df = pd.DataFrame({
    "torque": pd.Series([1, 2, 2, 3], dtype="pint[lbf ft]"),
    "angular_velocity": pd.Series([1, 2, 2, 3], dtype="pint[rpm]"),
  })

[2]: torque angular_velocity
     0   1   1
     1   2   2
     2   2   2
     3   3   3
```

Operations with columns are units aware so behave as we would intuitively expect.

```python
[3]: df['power'] = df['torque'] * df['angular_velocity']

[3]: torque angular_velocity power
     0   1   1   1
     1   2   2   4
     2   2   2   4
     3   3   3   9
```

We can see the columns’ units in the dtypes attribute

```python
[4]: df.dtypes
```

```python
traceback:...
pint['rpm']
```

(continues on next page)
Each column can be accessed as a Pandas Series:

```
df.power
```

Which contains a PintArray:

```
df.power.values
```

The PintArray contains a Quantity:

```
df.power.values.quantity
```

Pandas Series accessors are provided for most Quantity properties and methods, which will convert the result to a Series where possible:

```
df.power.pint.units
```

```
df.power.pint.to("kW").values
```

---

### 3.15.2 Reading from csv

Reading from files is the far more standard way to use pandas. To facilitate this, DataFrame accessors are provided to make it easy to get to PintArrays:

```
import pandas as pd
import pint
import io

Here's the contents of the csv file.
```

```
test_data = '''speed,mech power,torque,rail pressure,fuel flow rate,fluid power
rpm,kW,N m,bar,l/min,kW
1000.0,,10.0,1000.0,10.0,'''
```

Let's read that into a DataFrame. Here `io.StringIO` is used in place of reading a file from disk, whereas a `csv` file path would typically be used and is shown commented.

```
[12]: df = pd.read_csv(io.StringIO(test_data),header=[0,1])
# df = pd.read_csv("/path/to/test_data.csv",header=[0,1])
[12]: 
speed mech power torque rail pressure fuel flow rate fluid power
rpm kW N m bar l/min kW
0 1000.0 NaN 10.0 1000.0 10.0 NaN
1 1100.0 NaN 10.0 100000000.0 10.0 NaN
2 1200.0 NaN 10.0 1000.0 10.0 NaN
3 1200.0 NaN 10.0 1000.0 10.0 NaN
```

Then use the DataFrame’s pint accessor’s quantify method to convert the columns from `np.ndarray` to `PintArrays`, with units from the bottom column level.

```
[13]: df.dtypes
[13]: speed rpm float64
mech power kW float64
torque N m float64
rail pressure bar float64
fuel flow rate l/min float64
fluid power kW float64
dtype: object
[14]: df_ = df.pint.quantify(level=-1)
[14]: 
speed mech power torque rail pressure fuel flow rate fluid power
rpm kW N m bar l/min kW
0 1000.0 NaN 10.0 1000.0 10.0 NaN
1 1100.0 NaN 10.0 100000000.0 10.0 NaN
2 1200.0 NaN 10.0 1000.0 10.0 NaN
3 1200.0 NaN 10.0 1000.0 10.0 NaN
```

As previously, operations between DataFrame columns are unit aware

```
[15]: df_.speed*df_.torque
[15]: 0 10000.0
1 11000.0
2 12000.0
3 12000.0
dtype: pint[meter * newton * revolutions_per_minute]
[16]: df_ 
[16]: 
speed mech power torque rail pressure fuel flow rate fluid power
rpm kW N m bar l/min kW
0 1000.0 NaN 10.0 1000.0 10.0 NaN
1 1100.0 NaN 10.0 100000000.0 10.0 NaN
2 1200.0 NaN 10.0 1000.0 10.0 NaN
3 1200.0 NaN 10.0 1000.0 10.0 NaN
```
The DataFrame’s `pint.dequantify` method then allows us to retrieve the units information as a header row once again.

This allows for some rather powerful abilities. For example, to change single column units

The units are harder to read than they need be, so let’s change pint’s default format for displaying units.
or the entire table’s units

```
[21]: df_.pint.to_base_units().pint.dequantify()
```

```
<table>
<thead>
<tr>
<th>unit</th>
<th>speed</th>
<th>mech power</th>
<th>torque</th>
<th>rail pressure</th>
<th>fuel flow rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>rad/s</td>
<td>kg·m²/s³</td>
<td>kg·m²/s²</td>
<td>kg/m/s²</td>
<td>m³/s</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>104,719,755</td>
<td>1047,197,551</td>
<td>10</td>
<td>1.000000e+08</td>
<td>0.000167</td>
</tr>
<tr>
<td>1</td>
<td>115,191,731</td>
<td>1151,917,306</td>
<td>10</td>
<td>1.000000e+13</td>
<td>0.000167</td>
</tr>
<tr>
<td>2</td>
<td>125,663,706</td>
<td>1256,637,061</td>
<td>10</td>
<td>1.000000e+08</td>
<td>0.000167</td>
</tr>
<tr>
<td>3</td>
<td>125,663,706</td>
<td>1256,637,061</td>
<td>10</td>
<td>1.000000e+08</td>
<td>0.000167</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fluid power</th>
<th>unit</th>
<th>kg·m²/s³</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.666667e+04</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.666667e+09</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.666667e+04</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.666667e+04</td>
<td></td>
</tr>
</tbody>
</table>
```

### 3.15.3 Advanced example

This example shows alternative ways to use pint with pandas and other features.

Start with the same imports.

```
[22]: import pandas as pd
    import pint
```

We’ll be use a shorthand for PintArray

```
[23]: PA_ = pint.PintArray
```

And set up a unit registry and quantity shorthand.

```
[24]: ureg=pint.UnitRegistry()
    Q_=ureg.Quantity
```

Operations between PintArrays of different unit registry will not work. We can change the unit registry that will be used in creating new PintArrays to prevent this issue.

```
[25]: pint.PintType.ureg = ureg
```

These are the possible ways to create a PintArray.

Note that pint[unit] must be used for the Series constructor, whereas the PintArray constructor allows the unit string or object.

```
[26]: df = pd.DataFrame({
    "length" : pd.Series([1,2], dtype="pint[m"]),
    "width" : PA_([2,3], dtype="pint[m"]),
    "distance" : PA_([2,3], dtype="m"),
    "height" : PA_([2,3], dtype=ureg.m),
    "depth" : PA_.from_1darray_quantity(Q_([2,3],ureg.m)),
    "fluid power" : PA_([2,3], dtype=ureg.kg·m²/s³)
})
```

```
| length width distance height depth fluid power |
|-------|-------|-------|-------|-------|
| 0     | 1     | 2     | 2     | 2     | 2     |
| 1     | 2     | 3     | 3     | 3     | 3     |
```

### 3.15. Pandas support
[27]: df.length.values.units

[27]: meter
4.1 Developer reference

4.1.1 Pint

Pint is Python module/package to define, operate and manipulate physical quantities: the product of a numerical value and a unit of measurement. It allows arithmetic operations between them and conversions from and to different units.

Copyright 2016 by Pint Authors, see AUTHORS for more details.

License BSD, see LICENSE for more details.

class pint.Context (name=None, aliases=(), defaults=None)
A specialized container that defines transformation functions from one dimension to another. Each Dimension are specified using a UnitsContainer. Simple transformation are given with a function taking a single parameter.

>>> timedim = UnitsContainer({'[time]': 1})
>>> spacedim = UnitsContainer({'[length]': 1})
>>> def f(time):
...     'Time to length converter'
...     return 3. * time
>>> c = Context()
>>> c.add_transformation(timedim, spacedim, f)
6

Conversion functions may take optional keyword arguments and the context can have default values for these arguments.

>>> def f(time, n):
...     'Time to length converter, n is the index of refraction of the material'

(continues on next page)
... return 3. * time / n

```python
c = Context(n=3)
c.add_transformation(timedim, spacedim, f)
c.transform(timedim, spacedim, 2)
```

2

**add_transformation**(src, dst, func)
Add a transformation function to the context.

**classmethod from_context**(context, **defaults)**
Creates a new context that shares the funcs dictionary with the original context. The default values are copied from the original context and updated with the new defaults.

If defaults is empty, return the same context.

**remove_transformation**(src, dst)
Add a transformation function to the context.

**transform**(src, dst, registry, value)
Transform a value.

**class pint.Measurement**
Implements a class to describe a quantity with uncertainty.

**Parameters**

- **value**(pint.Quantity or any numeric type) – The expected value of the measurement
- **error**(pint.Quantity or any numeric type) – The error or uncertainty of the measurement

**class pint.Quantity**
Implements a class to describe a physical quantity: the product of a numerical value and a unit of measurement.

**Parameters**

- **value**(str, pint.Quantity or any numeric type) – value of the physical quantity to be created
- **units**(UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created

**check**(dimension)
Return true if the quantity’s dimension matches passed dimension.

**dimensionality**
Quantity’s dimensionality (e.g. {length: 1, time: -1})

**dimensionless**
Return true if the quantity is dimensionless.

**classmethod from_list**(quant_list, units=None)
Transforms a list of Quantities into an numpy.array quantity. If no units are specified, the unit of the first element will be used. Same as from_sequence.

If units is not specified and list is empty, the unit cannot be determined and a ValueError is raised.

**Parameters**

- **quant_list**(list of pint.Quantity) – list of pint.Quantity
• units (UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created

classmethod from_sequence (seq, units=None)
Transforms a sequence of Quantities into an numpy.array quantity. If no units are specified, the unit of the first element will be used.

If units is not specified and sequence is empty, the unit cannot be determined and a ValueError is raised.

Parameters

• seq (sequence of pint.Quantity) – sequence of pint.Quantity

• units (UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created

ito (other=None, *contexts, **ctx_kwargs)
Inplace rescale to different units.

Parameters other (pint.Quantity, str or dict) – destination units.

ito_base_units ()
Return Quantity rescaled to base units

ito_reduced_units ()
Return Quantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (intentionally), nor can it make use of contexts at this time.

ito_root_units ()
Return Quantity rescaled to base units

m
Quantity’s magnitude. Short form for magnitude

m_as (units)
Quantity’s magnitude expressed in particular units.

Parameters units (pint.Quantity, str or dict) – destination units

magnitude
Quantity’s magnitude. Long form for m

to (other=None, *contexts, **ctx_kwargs)
Return Quantity rescaled to different units.

Parameters other (pint.Quantity, str or dict) – destination units.

to_base_units ()
Return Quantity rescaled to base units

to_compact (unit=None)
Return Quantity rescaled to compact, human-readable units.

To get output in terms of a different unit, use the unit parameter.

```python
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> (200e-9 * ureg.s).to_compact()
<Quantity(200.0, 'nanosecond')>
>>> (1e-2 * ureg('kg m/s^2')).to_compact('N')
<Quantity(10.0, 'millinewton')>
```
to_reduced_units()
Return Quantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (intentionally), nor can it make use of contexts at this time.

to_root_units()
Return Quantity rescaled to base units

u
Quantity’s units. Short form for units

Return type UnitsContainer

unitless
Return true if the quantity does not have units.

units
Quantity’s units. Long form for u

Return type UnitsContainer

class pint.Unit(units)
Implements a class to describe a unit supporting math operations.

dimensionality
Unit’s dimensionality (e.g. {length: 1, time: -1})

dimensionless
Return true if the Unit is dimensionless.

from_(value, strict=True, name=u'value')
Converts a numerical value or quantity to this unit

Parameters
• value – a Quantity (or numerical value if strict=False) to convert
• strict – boolean to indicate that only quantities are accepted
• name – descriptive name to use if an exception occurs

Returns The converted value as this unit

Raises ValueError if strict and one of the arguments is not a Quantity.

m_from(value, strict=True, name=u'value')
Converts a numerical value or quantity to this unit, then returns the magnitude of the converted value

Parameters
• value – a Quantity (or numerical value if strict=False) to convert
• strict – boolean to indicate that only quantities are accepted
• name – descriptive name to use if an exception occurs

Returns The magnitude of the converted value

Raises ValueError if strict and one of the arguments is not a Quantity.

class pint.UnitRegistry(filename=u'', force_ndarray=False, default_as_delta=True, autoconvert_offset_to_baseunit=False, on_redefinition=u'warn', system=None, auto_reduce_dimensions=False)
The unit registry stores the definitions and relationships between units.

Parameters
• **filename** – path of the units definition file to load or line-iterable object. Empty to load the default definition file. None to leave the UnitRegistry empty.

• **force_ndarray** – convert any input, scalar or not to a numpy.ndarray.

• **default_as_delta** – In the context of a multiplication of units, interpret non-multiplicative units as their delta counterparts.

• **autoconvert_offset_to_baseunit** – If True converts offset units in quantities are converted to their base units in multiplicative context. If False no conversion happens.

• **on_redefinition**(str) – action to take in case a unit is redefined. ‘warn’, ‘raise’, ‘ignore’

• **auto_reduce_dimensions** – If True, reduce dimensionality on appropriate operations.

check (*args)

Decorator to for quantity type checking for function inputs.

Use it to ensure that the decorated function input parameters match the expected type of pint quantity.

Use None to skip argument checking.

Parameters

• **ureg** – a UnitRegistry instance.

• **args** – iterable of input units.

Returns the wrapped function.

Raises pint.DimensionalityError – if the parameters don’t match dimensions

pi_theorem (quantities)

Builds dimensionless quantities using the Buckingham \( \pi \) theorem.

:param quantities: mapping between variable name and units
:type quantities: dict
:return: a list of dimensionless quantities expressed as dicts

setup_matplotlib (enable=True)

Set up handlers for matplotlib’s unit support.

:param enable: whether support should be enabled or disabled
:type enable: bool

wraps (ret, args, strict=True)

Wraps a function to become pint-aware.

Use it when a function requires a numerical value but in some specific units. The wrapper function will take a pint quantity, convert to the units specified in args and then call the wrapped function with the resulting magnitude.

The value returned by the wrapped function will be converted to the units specified in ret.

Use None to skip argument conversion. Set strict to False, to accept also numerical values.

Parameters

• **ureg** – a UnitRegistry instance.

• **ret** – output units.

• **args** – iterable of input units.

• **strict** – boolean to indicate that only quantities are accepted.

Returns the wrapped function.

Raises ValueError if strict and one of the arguments is not a Quantity.
exception pint.DimensionalityError (units1, units2, dim1=None, dim2=None, extra_msg=u")
    Raised when trying to convert between incompatible units.

exception pint.OffsetUnitCalculusError (units1, units2=u", extra_msg=u")
    Raised on ambiguous operations with offset units.

exception pint.UndefineUnitError (unit_names)
    Raised when the units are not defined in the unit registry.

exception pint.UnitStrippedWarning

pint.get_application_registry()
    Return the application registry. If set_application_registry() was never invoked, return a registry
    built using defaults_en.txt embedded in the pint package.

    Parameters
    registry  a UnitRegistry instance.

pint.set_application_registry (registry)
    Set the application registry, which is used for unpickling operations and when invoking pint.Quantity or pint.Unit
    directly.

    Parameters
    registry  a UnitRegistry instance.

pint.babel

    copyright  2016 by Pint Authors, see AUTHORS for more details.
    license BSD, see LICENSE for more details.

pint.context

    Functions and classes related to context definitions and application.

    copyright  2016 by Pint Authors, see AUTHORS for more details.
    license BSD, see LICENSE for more details.

    class pint.context.Context (name=None, aliases=(), defaults= None)
        A specialized container that defines transformation functions from one dimension to another. Each Dimension
        are specified using a UnitsContainer. Simple transformation are given with a function taking a single parameter.

        >>> timedim = UnitsContainer({"[time]": 1})
        >>> spacedim = UnitsContainer({"[length]": 1})
        >>> def f(time):
        ...     'Time to length converter'
        ...     return 3. * time
        >>> c = Context()
        >>> c.add_transformation(timedim, spacedim, f)
        >>> c.transform(timedim, spacedim, 2)
        6

        Conversion functions may take optional keyword arguments and the context can have default values for these
        arguments.

        >>> def f(time, n):
        ...     'Time to length converter, n is the index of refraction of the material'
        ...     return 3. * time / n
        >>> c = Context(n=3)
        >>> c.add_transformation(timedim, spacedim, f)
>>> c.transform(timedim, spacedim, 2)

2

add_transformation (src, dst, func)
Add a transformation function to the context.

classmethod from_context (context, **defaults)
Creates a new context that shares the funcs dictionary with the original context. The default values are copied from the original context and updated with the new defaults.

If defaults is empty, return the same context.

remove_transformation (src, dst)
Add a transformation function to the context.

transform (src, dst, registry, value)
Transform a value.

class pint.context.ContextChain (*args, **kwargs)
A specialized ChainMap for contexts that simplifies finding rules to transform from one dimension to another.

graph
The graph relating

insert_contexts (*contexts)
Insert one or more contexts in reversed order the chained map. (A rule in last context will take precedence)

To facilitate the identification of the context with the matching rule, the relation_to_context dictionary of the context is used.

remove_contexts (n)
Remove the last n inserted contexts from the chain.

transform (src, dst, registry, value)
Transform the value, finding the rule in the chained context. (A rule in last context will take precedence)

Raises KeyError if the rule is not found.

class pint.converters.Converter
Base class for value converters.

class pint.converters.OffsetConverter (scale, offset)
An affine transformation

class pint.converters.ScaleConverter (scale)
A linear transformation

class pint.converters
Functions and classes related to unit conversions.

copyright 2016 by Pint Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

pint.definitions
Functions and classes related to unit definitions.

copyright 2016 by Pint Authors, see AUTHORS for more details.
class pint.definitions.AliasDefinition(name, aliases)
    Additional alias(es) for an already existing unit

class pint.definitions.Definition(name, symbol, aliases, converter)
    Base class for definitions.

        Parameters
        • name – name.
        • symbol – a short name or symbol for the definition
        • aliases – iterable of other names.
        • converter – an instance of Converter.

    classmethod from_string(definition)
        Parse a definition

class pint.definitions.DimensionDefinition(name, symbol, aliases, converter, reference=None, is_base=False)
    Definition of a dimension.

class pint.definitions.PrefixDefinition(name, symbol, aliases, converter)
    Definition of a prefix.

class pint.definitions.UnitDefinition(name, symbol, aliases, converter, reference=None, is_base=False)
    Definition of a unit.

        Parameters
        • reference – Units container with reference units.
        • is_base – indicates if it is a base unit.

pint.errors

Functions and classes related to unit definitions and conversions.

copyright 2016 by Pint Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

exception pint.errors.DefinitionSyntaxError(msg, filename=None, lineno=None)
    Raised when a textual definition has a syntax error.

exception pint.errors.DimensionalityError(units1, units2, dim1=None, dim2=None, extra_msg=u"
    Raised when trying to convert between incompatible units.

exception pint.errors.OffsetUnitCalculusError(units1, units2=u", extra_msg=u"
    Raised on ambiguous operations with offset units.

exception pint.errors.RedefinitionError(name, definition_type)
    Raised when a unit or prefix is redefined.

exception pint.errors.UndefinedUnitError(unit_names)
    Raised when the units are not defined in the unit registry.

exception pint.errors.UnitStrippedWarning
**pint.formatter**

Format units for pint.

```python
pint.formatter(items, as_ratio=True, single_denominator=False, product_fmt=u' * ', division_fmt=u' / ', power_fmt=u'**', parentheses_fmt=u'({0})', exp_call=<function <lambda>>, locale=None, babel_length=u'long', babel_plural_form=u'one')
```

Format a list of (name, exponent) pairs.

**Parameters**

- `items` – a list of (name, exponent) pairs.
- `as_ratio` – True to display as ratio, False as negative powers.
- `single_denominator` – all with terms with negative exponents are collected together.
- `product_fmt` – the format used for multiplication.
- `division_fmt` – the format used for division.
- `power_fmt` – the format used for exponentiation.
- `parentheses_fmt` – the format used for parenthesis.
- `locale` – the locale object as defined in babel.
- `locale` – the length of the translated unit, as defined in babel cldr.
- `babel_plural_form` – the plural form, calculated as defined in babel.

**Returns** the formula as a string.

```python
pint.formatting.siunitx_format_unit(units)
```

Returns LaTeX code for the unit that can be put into an siunitx command.

**pint.matplotlib**

Functions and classes related to working with Matplotlib’s support for plotting with units.

```python
class pint.matplotlib.PintAxisInfo(units)
  Support default axis and tick labeling and default limits.

class pint.matplotlib.PintConverter(registry)
  Implement support for pint within matplotlib’s unit conversion framework.

static axisinfo(unit, axis)
  Return axis information for this particular unit.

convert(value, unit, axis)
  Convert :Quantity instances for matplotlib to use.

static default_units(x, axis)
  Get the default unit to use for the given combination of unit and axis.
```

4.1. Developer reference
```
pint.matplotlib.setup_matplotlib_handlers(registry, enable)

Set up matplotlib's unit support to handle units from a registry.
:param registry: the registry that will be used
:type registry: UnitRegistry
:param enable: whether support should be enabled or disabled
:type enable: bool
```

```
class pint.measurement.Measurement

Implements a class to describe a quantity with uncertainty.

Parameters

- **value** (pint.Quantity or any numeric type) – The expected value of the measurement
- **error** (pint.Quantity or any numeric type) – The error or uncertainty of the measurement
```

```
pint.eval

An expression evaluator to be used as a safe replacement for builtin eval.

```
```
class pint.quantity.Quantity

Implements a class to describe a physical quantity: the product of a numerical value and a unit of measurement.

Parameters

- **value** (str, pint.Quantity or any numeric type) – value of the physical quantity to be created
- **units** (UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created
```
check \( (\text{dimension}) \)
Return true if the quantity’s dimension matches passed dimension.

dimensionality
Quantity’s dimensionality (e.g. \{length: 1, time: -1\})
dimensionless
Return true if the quantity is dimensionless.

classmethod from_list \( (\text{quant\_list}, \text{units}=\text{None}) \)
Transforms a list of Quantities into an numpy.array quantity. If no units are specified, the unit of the first element will be used. Same as from_sequence.

If units is not specified and list is empty, the unit cannot be determined and a ValueError is raised.

Parameters

- \text{quant\_list} (list of pint.Quantity) – list of pint.Quantity
- \text{units} (UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created

classmethod from_sequence \( (\text{seq}, \text{units}=\text{None}) \)
Transforms a sequence of Quantities into an numpy.array quantity. If no units are specified, the unit of the first element will be used.

If units is not specified and sequence is empty, the unit cannot be determined and a ValueError is raised.

Parameters

- \text{seq} (sequence of pint.Quantity) – sequence of pint.Quantity
- \text{units} (UnitsContainer, str or pint.Quantity) – units of the physical quantity to be created

ito \( (\text{other}=\text{None}, \ast\text{contexts}, **\text{ctx\_kwargs}) \)
Inplace rescale to different units.

Parameters

- \text{other} (pint.Quantity, str or dict) – destination units.

ito_base_units
Return Quantity rescaled to base units

ito_reduced_units
Return Quantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (intentionally), nor can it make use of contexts at this time.

ito_root_units
Return Quantity rescaled to base units

m
Quantity’s magnitude. Short form for magnitude

m\_as \( (\text{units}) \)
Quantity’s magnitude expressed in particular units.

Parameters

- \text{units} (pint.Quantity, str or dict) – destination units

magnitude
Quantity’s magnitude. Long form for \( m \)

to \( (\text{other}=\text{None}, \ast\text{contexts}, **\text{ctx\_kwargs}) \)
Return Quantity rescaled to different units.

Parameters

- \text{other} (pint.Quantity, str or dict) – destination units.
to_base_units()
Return Quantity rescaled to base units

to_compact(unit=None)
Return Quantity rescaled to compact, human-readable units.

To get output in terms of a different unit, use the unit parameter.

```python
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> (200e-9*ureg.s).to_compact()
<Quantity(200.0, 'nanosecond')>
>>> (1e-2*ureg('kg m/s^2')).to_compact('N')
<Quantity(10.0, 'millinewton')>
```

to_reduced_units()
Return Quantity scaled in place to reduced units, i.e. one unit per dimension. This will not reduce compound units (intentionally), nor can it make use of contexts at this time.

to_root_units()
Return Quantity rescaled to base units

u
Quantity’s units. Short form for units

Return type UnitsContainer

unitless
Return true if the quantity does not have units.

units
Quantity’s units. Long form for u

Return type UnitsContainer

pint.quantity.printoptions(*args, **kwds)
Numpy printoptions context manager released with version 1.15.0 https://docs.scipy.org/doc/numpy/reference/generated/numpy.printoptions.html

pint.registry
Defines the Registry, a class to contain units and their relations.

The module actually defines 5 registries with different capabilities:

- **BaseRegistry**: Basic unit definition and querying. Conversion between multiplicative units.
- **NonMultiplicativeRegistry**: Conversion between non multiplicative (offset) units. (e.g. Temperature)
  - Inherits from BaseRegistry
- **ContextRegistry**: Conversion between units with different dimensions according to previously established relations (contexts). (e.g. in the spectroscopy, conversion between frequency and energy is possible)
  - Inherits from BaseRegistry
- **SystemRegistry**: Group unit and changing of base units. (e.g. in MKS, meter, kilogram and second are base units.)
  - Inherits from BaseRegistry
- **UnitRegistry**: Combine all previous capabilities, it is exposed by Pint.
class `pint.registry.BaseRegistry` (filename=u"", force_ndarray=False,
                         on_redefinition=u"warn", auto_reduce_dimensions=False)

Base class for all registries.

Capabilities: - Register units, prefixes, and dimensions, and their relations. - Convert between units. - Find
dimensionality of a unit. - Parse units with prefix and/or suffix. - Parse expressions. - Parse a definition file. -
Allow extending the definition file parser by registering @ directives.

Parameters

- `filename (str or None)`: path of the units definition file to load or line iterable
  object. Empty to load the default definition file. None to leave the UnitRegistry empty.
- `force_ndarray`: convert any input, scalar or not to a numpy.ndarray.
- `on_redefinition (str)`: action to take in case a unit is redefined. ‘warn’, ‘raise’,
  ‘ignore’
- `auto_reduce_dimensions`: If True, reduce dimensionality on appropriate opera-
tions.

`convert (value, src, dst, inplace=False)`

Convert value from some source to destination units.

Parameters

- `value`: value
- `src (pint.Quantity or str)`: source units.
- `dst (pint.Quantity or str)`: destination units.

Returns converted value

`default_format`

Default formatting string for quantities.

`define (definition)`

Add unit to the registry.

Parameters `definition (str or Definition)`: a dimension, unit or prefix definition.

`get_base_units (input_units, check_nonmult=True, system=None)`

Convert unit or dict of units to the base units.

If any unit is non multiplicative and check_converter is True, then None is returned as the multiplicative
factor.

Parameters

- `input_units (UnitsContainer or str)`: units
- `check_nonmult`: if True, None will be returned as the multiplicative factor if a non-
multiplicative units is found in the final Units.

Returns multiplicative factor, base units

`get_compatible_units (input_units, group_or_system=None)`

`get_dimensionality (input_units)`

Convert unit or dict of units or dimensions to a dict of base dimensions dimensions

Parameters `input_units` -
Returns  dimensionality

get_name  (name_or_alias, case_sensitive=True)
Return the canonical name of a unit.

get_root_units  (input_units, check_nonmult=True)
Convert unit or dict of units to the root units.

If any unit is non multiplicative and check_converter is True, then None is returned as the multiplicative factor.

Parameters

•  input_units  (UnitsContainer  or  str) – units

•  check_nonmult  – if True, None will be returned as the multiplicative factor if a non-
multiplicative units is found in the final Units.

Returns  multiplicative factor, base units

get_symbol  (name_or_alias)
Return the preferred alias for a unit

load_definitions  (file, is_resource=False)
Add units and prefixes defined in a definition text file.

Parameters

•  file – can be a filename or a line iterable.

•  is_resource – used to indicate that the file is a resource file and therefore should be
loaded from the package.

parse_expression  (input_string, case_sensitive=True, use_decimal=False, **values)
Parse a mathematical expression including units and return a quantity object.

Numerical constants can be specified as keyword arguments and will take precedence over the names
defined in the registry.

parse_unit_name  (unit_name, case_sensitive=True)
Parse a unit to identify prefix, unit name and suffix by walking the list of prefix and suffix.

Return type  (str, str, str)

parse_units  (input_string, as_delta=None)
Parse a units expression and returns a UnitContainer with the canonical names.

The expression can only contain products, ratios and powers of units.

Parameters as_delta  – if the expression has multiple units, the parser will interpret non
multiplicative units as their delta_ counterparts.

Raises pint.UndefindUnitError  if a unit is not in the registry ValueError  if the
expression is invalid.

class  pint.registry.ContextRegistry(**kwargs)
Handle of Contexts.
Conversion between units with different dimensions according to previously established relations (contexts).
(e.g. in the spectroscopy, conversion between frequency and energy is possible)

Capabilities: - Register contexts. - Enable and disable contexts. - Parse @context directive.

add_context  (context)
Add a context object to the registry.
The context will be accessible by its name and aliases.
Notice that this method will NOT enable the context. Use `enable_contexts`.

`context (**kwds)`
Used as a context manager, this function enables to activate a context which is removed after usage.

**Parameters**

- `names` – name of the context.
- `kwargs` – keyword arguments for the contexts.

Context are called by their name:

```python
>>> with ureg.context('one):
...     pass
```

If the context has an argument, you can specify its value as a keyword argument:

```python
>>> with ureg.context('one', n=1):
...     pass
```

Multiple contexts can be entered in single call:

```python
>>> with ureg.context('one', 'two', n=1):
...     pass
```

or nested allowing you to give different values to the same keyword argument:

```python
>>> with ureg.context('one', n=1):
...     with ureg.context('two', n=2):
...         pass
```

A nested context inherits the defaults from the containing context:

```python
>>> with ureg.context('one', n=1):
...     with ureg.context('two): # Here n takes the value of the upper context
...         pass
```

`disable_contexts (n=None)`
Disable the last n enabled contexts.

`enable_contexts (*names_or_contexts, **kwargs)`
Enable contexts provided by name or by object.

**Parameters**

- `names_or_contexts` – sequence of the contexts or contexts names/alias
- `kwargs` – keyword arguments for the context

`remove_context (name_or_alias)`
Remove a context from the registry and return it.

Notice that this methods will not disable the context. Use `disable_contexts`.

`with_context (name, **kw)`
Decorator to wrap a function call in a Pint context.

Use it to ensure that a certain context is active when calling a function:
```python
>>> @ureg.with_context('sp')
... def my_cool_fun(wavelenght):
...     print('This wavelength is equivalent to: %s', wavelength.to('terahertz'))
```
**get_system** *(name, create_if_needed=True)*

Return a Group.

**Parameters**

- **name** – Name of the group to be
- **create_if_needed** – Create a group if not Found. If False, raise an Exception.

**Returns** System

```python
class pint.registry.UnitRegistry(filename=u'', force_ndarray=False, default_as_delta=True, autoconvert_offset_to_baseunit=False, on_redefinition=u'warn', auto_reduce_dimensions=False, system=None)
```

The unit registry stores the definitions and relationships between units.

**Parameters**

- **filename** – path of the units definition file to load or line-iterable object. Empty to load the default definition file. None to leave the UnitRegistry empty.
- **force_ndarray** – convert any input, scalar or not to a numpy.ndarray.
- **default_as_delta** – In the context of a multiplication of units, interpret non-multiplicative units as their delta counterparts.
- **autoconvert_offset_to_baseunit** – If True converts offset units in quantities are converted to their base units in multiplicative context. If False no conversion happens.
- **on_redefinition** *(str)* – action to take in case a unit is redefined. ‘warn’, ‘raise’, ‘ignore’
- **auto_reduce_dimensions** – If True, reduce dimensionality on appropriate operations.

**check** *(args)*

Decorator to for quantity type checking for function inputs.

Use it to ensure that the decorated function input parameters match the expected type of pint quantity.

Use None to skip argument checking.

**Parameters**

- **ureg** – a UnitRegistry instance.
- **args** – iterable of input units.

**Returns** the wrapped function.

**Raises** pint.DimensionalityError – if the parameters don’t match dimensions

**pi_theorem** *(quantities)*

Builds dimensionless quantities using the Buckingham \( \pi \) theorem

**:param quantities:** mapping between variable name and units
**:type quantities:** dict
**:return:** a list of dimensionless quantities expressed as dicts

**setup_matplotlib** *(enable=True)*

Set up handlers for matplotlib’s unit support.

**:param enable:** whether support should be enabled or disabled
**:type enable:** bool

**wraps** *(ret, args, strict=True)*

Wraps a function to become pint-aware.
Use it when a function requires a numerical value but in some specific units. The wrapper function will take a pint quantity, convert to the units specified in \texttt{args} and then call the wrapped function with the resulting magnitude.

The value returned by the wrapped function will be converted to the units specified in \texttt{ret}.

Use None to skip argument conversion. Set strict to False, to accept also numerical values.

\textbf{Parameters}

- \texttt{ureg} – a UnitRegistry instance.
- \texttt{ret} – output units.
- \texttt{args} – iterable of input units.
- \texttt{strict} – boolean to indicate that only quantities are accepted.

\textbf{Returns} the wrapped function.

\textbf{Raises} \texttt{ValueError} if strict and one of the arguments is not a Quantity.

\texttt{pint.registry_helpers}

Miscellaneous methods of the registry written as separate functions.

\textbf{copyright} 2016 by Pint Authors, see AUTHORS for more details..

\textbf{license} BSD, see LICENSE for more details.

\texttt{pint.registry_helpers.check (ureg, *args)}

Decorator to for quantity type checking for function inputs.

Use it to ensure that the decorated function input parameters match the expected type of pint quantity.

Use None to skip argument checking.

\textbf{Parameters}

- \texttt{ureg} – a UnitRegistry instance.
- \texttt{args} – iterable of input units.

\textbf{Returns} the wrapped function.

\textbf{Raises} \texttt{pint.DimensionalityError} – if the parameters don’t match dimensions

\texttt{pint.registry_helpers.wraps (ureg, ret, args, strict=True)}

Wraps a function to become pint-aware.

Use it when a function requires a numerical value but in some specific units. The wrapper function will take a pint quantity, convert to the units specified in \texttt{args} and then call the wrapped function with the resulting magnitude.

The value returned by the wrapped function will be converted to the units specified in \texttt{ret}.

Use None to skip argument conversion. Set strict to False, to accept also numerical values.

\textbf{Parameters}

- \texttt{ureg} – a UnitRegistry instance.
- \texttt{ret} – output units.
- \texttt{args} – iterable of input units.
- \texttt{strict} – boolean to indicate that only quantities are accepted.
Returns the wrapped function.

Raises `ValueError` if strict and one of the arguments is not a Quantity.

```python
pint.systems
```

Functions and classes related to system definitions and conversions.

```python
class pint.systems.Group(name)
```

A group is a set of units.

Units can be added directly or by including other groups.

Members are computed dynamically, that is if a unit is added to a group X all groups that include X are affected.

The group belongs to one Registry.

It can be specified in the definition file as:

```python
@group <name> [using <group 1>, ..., <group N>]
  <definition 1>
  ...
  <definition N>
@end
```

```python
add_groups(*group_names)
```

Add groups to group.

```python
add_units(*unit_names)
```

Add units to group.

```python
classmethod from_lines(lines, define_func)
```

Return a Group object parsing an iterable of lines.

Parameters

- **lines** (list[str]) – iterable
- **define_func**(str -> None) – Function to define a unit in the registry.

```python
invalidate_members()
```

Invalidate computed members in this Group and all parent nodes.

```python
members
```

Names of the units that are members of the group.

Calculated to include to all units in all included _used_groups.

Return type frozenset[str]

```python
remove_groups(*group_names)
```

Remove groups from group.

```python
remove_units(*unit_names)
```

Remove units from group.
class pint.systems.System(name)
A system is a Group plus a set of base units.

Members are computed dynamically, that is if a unit is added to a group X all groups that include X are affected.

The System belongs to one Registry.

It can be specified in the definition file as:

```text
@system <name> [using <group 1>, ..., <group N>]
  <rule 1>
  ...
  <rule N>
@end
```

The syntax for the rule is:

```text
new_unit_name : old_unit_name
```

where:

- old_unit_name: a root unit part which is going to be removed from the system.
- new_unit_name: a non root unit which is going to replace the old_unit.

If the new_unit_name and the old_unit_name, the later and the colon can be ommited.

```python
add_groups (*group_names)
```
Add groups to group.

```python
format_babel (locale)
```
Translate the name of the system

```python
invalidate_members()
```
Invalidate computed members in this Group and all parent nodes.

```python
remove_groups (*group_names)
```
Remove groups from group.

pint.unit

Functions and classes related to unit definitions and conversions.

```python
copyright 2016 by Pint Authors, see AUTHORS for more details.
```

```python
license BSD, see LICENSE for more details.
```

```python
class pint.unit.Unit (units)
```
Implements a class to describe a unit supporting math operations.

```python
dimensionality
  Unit’s dimensionality (e.g. {length: 1, time: -1})
```
\textbf{dimensionless}
Return true if the Unit is dimensionless.

\textbf{from\_} \texttt{(value, strict=True, name=u'value')}
Converts a numerical value or quantity to this unit

\hspace{1em} \textbf{Parameters}
\begin{itemize}
  \item \texttt{value} – a Quantity (or numerical value if strict=False) to convert
  \item \texttt{strict} – boolean to indicate that only quantities are accepted
  \item \texttt{name} – descriptive name to use if an exception occurs
\end{itemize}

\hspace{1em} \textbf{Returns} The converted value as this unit

\hspace{1em} \textbf{Raises} \texttt{ValueError} if strict and one of the arguments is not a Quantity.

\textbf{m\_from\_} \texttt{(value, strict=True, name=u'value')}
Converts a numerical value or quantity to this unit, then returns the magnitude of the converted value

\hspace{1em} \textbf{Parameters}
\begin{itemize}
  \item \texttt{value} – a Quantity (or numerical value if strict=False) to convert
  \item \texttt{strict} – boolean to indicate that only quantities are accepted
  \item \texttt{name} – descriptive name to use if an exception occurs
\end{itemize}

\hspace{1em} \textbf{Returns} The magnitude of the converted value

\hspace{1em} \textbf{Raises} \texttt{ValueError} if strict and one of the arguments is not a Quantity.

\texttt{pint.util}
Miscellaneous functions for \texttt{pint}.

\hspace{1em} \textbf{copyright} 2016 by Pint Authors, see AUTHORS for more details.

\hspace{1em} \textbf{license} BSD, see LICENSE for more details.

\texttt{class pint.util.BlockIterator}
Like SourceIterator but stops when it finds `@end` It also raises an error if another `@` directive is found inside.

\texttt{class pint.util.ParserHelper (scale=1, *args, **kwargs)}
The ParserHelper stores in place the product of variables and their respective exponent and implements the corresponding operations.

ParserHelper is a read-only mapping. All operations (even in place ones) return new instances.

WARNING : The hash value used does not take into account the scale attribute so be careful if you use it as a dict key and then two unequal object can have the same hash.

\texttt{classmethod from\_word (input\_word)}
Creates a ParserHelper object with a single variable with exponent one.

Equivalent to: \texttt{ParserHelper({'word': 1})}

\texttt{class pint.util.PrettyIPython}
Mixin to add pretty-printers for IPython

\texttt{class pint.util.SharedRegistryObject}
Base class for object keeping a reference to the registree.

Such object are for now Quantity and Unit, in a number of places it is that an object from this class has a `._units` attribute.
class pint.util.SourceIterator
    Iterator to facilitate reading the definition files.
    Accepts any sequence (like a list of lines, a file or another SourceIterator)
    The iterator yields the line number and line (skipping comments and empty lines) and stripping white spaces.
    for lineno, line in SourceIterator(sequence): # do something here
block_iter()
    Iterate block including header.

class pint.util.UnitsContainer(*args, **kwargs)
    The UnitsContainer stores the product of units and their respective exponent and implements the corresponding operations.
    UnitsContainer is a read-only mapping. All operations (even in place ones) return new instances.
    remove(keys)
        Create a new UnitsContainer purged from given keys.
    rename(oldkey, newkey)
        Create a new UnitsContainer in which an entry has been renamed.

pint.util.column_echelon_form(matrix, ntype=<class 'fractions.Fraction'>, transpose_result=False)
    Calculates the column echelon form using Gaussian elimination.

    Parameters
    • matrix – a 2D matrix as nested list.
    • ntype – the numerical type to use in the calculation.
    • transpose_result – indicates if the returned matrix should be transposed.

    Returns column echelon form, transformed identity matrix, swapped rows

pint.util.fix_str_conversions(cls)
    Enable python2/3 compatible behaviour for __str__.

pint.util.getattr_maybe_raise(self, item)
    Helper function to invoke at the beginning of all overridden __getattr__ methods. Raise AttributeError if
    the user tries to ask for a _ or __ attribute.

pint.util.infer_base_unit(q)
    Return UnitsContainer of q with all prefixes stripped.

pint.util.iterable(y)
    Check whether or not an object can be iterated over.
    Vendored from numpy under the terms of the BSD 3-Clause License. (Copyright (c) 2005-2019, NumPy Developers.)

    Parameters
    • value – Input object.
    • type – object

pint.util.matrix_to_string(matrix, row_headers=None, col_headers=None, fmtfun=<function <lambda>>)
    Takes a 2D matrix (as nested list) and returns a string.

pint.util.pi_theorem(quantities, registry=None)
    Builds dimensionless quantities using the Buckingham π theorem
**Parameters** quantities *(dict)* – mapping between variable name and units

**Returns** a list of dimensionless quantities expressed as dicts

```
import pint

pint.util.sized(y)
```

Check whether or not an object has a defined length.

**Parameters**

- **value** – Input object.
- **type** – object

```
pint.util.solve_dependencies(dependencies)
```

Solve a dependency graph.

**Parameters** dependencies – dependency dictionary. For each key, the value is an iterable indicating its dependencies.

**Returns** list of sets, each containing keys of independents tasks dependent only of the previous tasks in the list.

```
pint.util.to_units_container(unit_like, registry=None)
```

Convert a unit compatible type to a UnitsContainer.

```
pint.util.transpose(matrix)
```

Takes a 2D matrix (as nested list) and returns the transposed version.

```
class pint.util.udict
```

Custom dict implementing `__missing__`.

```
copy() → a shallow copy of D
```

**pint.compat.chainmap**

Taken from the Python 3.3 source code.

```
copyright 2013, PSF
license PSF License
```

```
class pint.compat.chainmap.ChainMap(*maps)
```

A ChainMap groups multiple dicts (or other mappings) together to create a single, updateable view.

The underlying mappings are stored in a list. That list is public and can accessed or updated using the `maps` attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

```
clear()
```

Clear maps[0], leaving maps[1:] intact.

```
copy()
```

New ChainMap or subclass with a new copy of maps[0] and refs to maps[1:]

```
classmethod fromkeys(iterable, *args)
```

Create a ChainMap with a single dict created from the iterable.

```
get(k[, d]) → D[k] if k in D, else d. d defaults to None.
```

```
new_child(m=None)
```

New ChainMap with a new map followed by all previous maps. If no map is provided, an empty dict is used.
parents

New ChainMap from maps[1:].

pop(key, *args)

Remove key from maps[0] and return its value. Raise KeyError if key not in maps[0].

popitem()

Remove and return an item pair from maps[0]. Raise KeyError is maps[0] is empty.

*pint.compatibility.lru_cache*

LRU (least recently used) cache backport.


copyright 2004, Raymond Hettinger,

license MIT License

pint.compat.lru_cache.lru_cache(maxsize=100, typed=False)

Least-recently-used cache decorator.

If maxsize is set to None, the LRU features are disabled and the cache can grow without bound.

If typed is True, arguments of different types will be cached separately. For example, f(3.0) and f(3) will be treated as distinct calls with distinct results.

Arguments to the cached function must be hashable.

View the cache statistics named tuple (hits, misses, maxsize, currsize) with f.cache_info(). Clear the cache and statistics with f.cache_clear(). Access the underlying function with f.__wrapped__.

See: http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

*pint.compatibility.meta*

Compatibility layer.

copyright 2016 by Pint Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

pint.compat.meta.with_metaclass(meta, *bases)

Create a base class with a metaclass.

Tokenization help for Python programs.

tokenize(readline) is a generator that breaks a stream of bytes into Python tokens. It decodes the bytes according to PEP-0263 for determining source file encoding.

It accepts a readline-like method which is called repeatedly to get the next line of input (or b’’’ for EOF). It generates 5-tuples with these members:

the token type (see token.py) the token (a string) the starting (row, column) indices of the token (a 2-tuple of ints) the ending (row, column) indices of the token (a 2-tuple of ints) the original line (string)

It is designed to match the working of the Python tokenizer exactly, except that it produces COMMENT tokens for comments and gives type OP for all operators. Additionally, all token lists start with an ENCODING token which tells you which encoding was used to decode the bytes stream.

exception pint.compat.tokenize.StopTokenizing

exception pint.compat.tokenize.TokenError
**pint Documentation, Release 0.10.dev0**

```python
class pint.compat.tokenize.TokenInfo
pint.compat.tokenize.detect_encoding(readline)
    The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, `readline`, in the same way as the `tokenize()` generator.

    It will call `readline` a maximum of twice, and return the encoding used (as a string) and a list of any lines (left as bytes) it has read in.

    It detects the encoding from the presence of a utf-8 bom or an encoding cookie as specified in pep-0263. If both a bom and a cookie are present, but disagree, a SyntaxError will be raised. If the encoding cookie is an invalid charset, raise a SyntaxError. Note that if a utf-8 bom is found, ‘utf-8-sig’ is returned.

    If no encoding is specified, then the default of ‘utf-8’ will be returned.
```

```python
pint.compat.tokenize.open(filename)
    Open a file in read only mode using the encoding detected by `detect_encoding()`.
```

```python
pint.compat.tokenize.tokenize(readline)
    The `tokenize()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects. Each call to the function should return one line of input as bytes. Alternately, `readline` can be a callable function terminating with `StopIteration`:

    ```python
    readline = open(myfile, 'rb').__next__  # Example of alternate readline
    ```

    The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the logical line; continuation lines are included.

    The first token sequence will always be an ENCODING token which tells you which encoding was used to decode the bytes stream.
```

```python
pint.compat.tokenize.untokenize(iterable)
    Transform tokens back into Python source code. It returns a bytes object, encoded using the ENCODING token, which is the first token sequence output by `tokenize`.

    Each element returned by the iterable must be a token sequence with at least two elements, a token number and token value. If only two tokens are passed, the resulting output is poor.
```

**Round-trip invariant for full input:** Untokenized source will match input source exactly

**Round-trip invariant for limited input:** # Output bytes will tokenize the back to the input $t1 = \{tok[:2] \mid tok \in tokenize(f.readline)\}$

    ```python
    newcode = untokenize(t1) readline = BytesIO(newcode).readline
t2 = \{tok[:2] \mid tok \in tokenize(readline)\}
    ```

    assert $t1 == t2$

```
class pint.testsuite.test_infer_base_unit.TestInferBaseUnit (methodName='runTest')
class pint.testsuite.test_issues.TestIssues (methodName='runTest')
class pint.testsuite.test_measurement.TestMeasurement (methodName='runTest')
class pint.testsuite.test_measurement.TestNotMeasurement (methodName='runTest')
class pint.testsuite.test_numpy.TestNDArrayQuantityMath (methodName='runTest')
class pint.testsuite.test_numpy.TestNumpyMethods (methodName='runTest')

classmethod setUpClass ()
    Hook method for setting up class fixture before running tests in the class.

test_reversible_op ()
test_searchsorted_numpy_func ()
    Test searchsorted as numpy function.

class pint.testsuite.test_numpy.TestNumpyNeedsSubclassing (methodName='runTest')

test_cross (**kwargs)
    Units are erased by asarray, Quantity does not inherit from NDArray

test_diff (**kwargs)
    Units are erased by asarray, Quantity does not inherit from NDArray

test_ediff1d (**kwargs)
    Units are erased by asarray, Quantity does not inherit from NDArray

test_fix (**kwargs)
    Units are erased by asarray, Quantity does not inherit from NDArray

test_gradient (**kwargs)
    shape is a property not a function

test_ones_like (**kwargs)
    Units are erased by emptyarra, Quantity does not inherit from NDArray

test_power (**kwargs)
    This is not supported as different elements might end up with different units
    eg. ([1, 1] * m) ** [2, 3]
    Must force exponent to single value

test_trapz (**kwargs)
    Units are erased by asanyarray, Quantity does not inherit from NDArray

test_unwrap (**kwargs)
    unwrap depends on diff

class pint.testsuite.test_pint_eval.TestPintEval (methodName='runTest')
class pint.testsuite.test_pitheorem.TestPiTheorem (methodName='runTest')
class pint.testsuite.test_quantity.TestCompareZero (methodName='runTest')
    This test case checks the special treatment that the zero value receives in the comparisons: pint>=0.9 supports comparisons against zero even for non-dimensionless quantities

class pint.testsuite.test_quantity.TestDimensionReduction (methodName='runTest')
class pint.testsuite.test_quantity.TestDimensions (methodName='runTest')
class pint.testsuite.test_quantity.TestDimensionsWithDefaultRegistry

    classmethod setUpClass()
    
        Hook method for setting up class fixture before running tests in the class.

class pint.testsuite.test_quantity.TestOffsetUnitMath

    test_addition_00001()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'kelvin')); expected_output = (110, u'kelvin')]

    test_addition_00002()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = error]

    test_addition_00003()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degF')); expected_output = error]

    test_addition_00004()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (105.56, u'kelvin')]

    test_addition_00005()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (110, u'kelvin')]

    test_addition_00006()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = (110, u'kelvin')]

    test_addition_00007()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = error]

    test_addition_00008()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

    test_addition_00009()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = error]

    test_addition_00010()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degR')); expected_output = error]

    test_addition_00011()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degC')); expected_output = (110, u'degC')]

    test_addition_00012()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degF')); expected_output = (105.56, u'degC')]

    test_addition_00013()
        test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'kelvin')); expected_output = error]
test_addition_00014()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degC')); expected_output = error]

test_addition_00015()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degF')); expected_output = error]

test_addition_00016()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degR')); expected_output = error]

test_addition_00017()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degC')); expected_output = (118, u'degF')]

test_addition_00018()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degF')); expected_output = (110, u'degF')]

test_addition_00019()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'kelvin')); expected_output = (118, u'degR')]

test_addition_00020()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degC')); expected_output = error]

test_addition_00021()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degF')); expected_output = error]

test_addition_00022()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected_output = (110, u'degR')]

test_addition_00023()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected_output = (118, u'degR')]

test_addition_00024()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (110, u'kelvin')]

test_addition_00025()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (110, u'kelvin')]

test_addition_00026()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = (110, u'degC')]

test_addition_00027()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degF')); expected_output = (190, u'degF')]

test_addition_00028()
test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degR')); expected_output = (190, u'degR')]
test_addition_00029()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degC')); expected_output = (110, u'delta_degC')]

test_addition_00030()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degF')); expected_output = (105.56, u'delta_degC')]

test_addition_00031()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'kelvin')); expected_output = (65.56, u'kelvin')]

test_addition_00032()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = (65.56, u'degC')]

test_addition_00033()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = (110, u'degF')]

test_addition_00034()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degR')); expected_output = (110, u'degR')]

test_addition_00035()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degC')); expected_output = (118, u'delta_degF')]

test_addition_00036()
  test_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degF')); expected_output = (110, u'delta_degF')]

test_division_with_scalar_00001()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'kelvin'), 2); expected_output = [(5.0, u'kelvin'), (5.0, u'kelvin')]]

test_division_with_scalar_00002()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'kelvin**2'), 2); expected_output = [(5.0, u'kelvin**2'), (5.0, u'kelvin**2')]]

test_division_with_scalar_00003()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), 2); expected_output = [u'error', u'error']]

test_division_with_scalar_00004()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC**2'), 2); expected_output = [u'error', u'error']]

test_division_with_scalar_00005()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC**-2'), 2); expected_output = [u'error', u'error']]

test_division_with_scalar_00006()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (10, u'kelvin')); expected_output = [(0.2, u'1/kelvin'), (0.2, u'1/kelvin')]]

test_division_with_scalar_00007()
  test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (10, u'degC')); expected_output = [u'error', (0.007063393960798164, u'1/kelvin')]]
test_division_with_scalar_00008()
    test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (10, u’degC**2'))]; expected_output = [u’error’, u’error’]

test_division_with_scalar_00009()
    test_division_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (10, u’degC**-2'))]; expected_output = [u’error’, u’error’]

test_exponentiation_00001()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), 1); expected_output = [(10, u’degC’), (10, u’degC’)]

test_exponentiation_00002()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), 0.5); expected_output = [(1.0, u’’), (1.0, u’’)]

test_exponentiation_00003()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), 0); expected_output = [(100.0, u’kelvin**2’), (100.0, u’kelvin**2’)]

test_exponentiation_00004()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), -1); expected_output = [(1.4142135623730951, 1.4142135623730951)]

test_exponentiation_00005()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), -2); expected_output = [(1.0, u’’), (1.0, u’’)]

test_exponentiation_00006()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((0, u’degC’), -2); expected_output = [(1.3402863367625568e-05, u’kelvin**-2’)]

test_exponentiation_00007()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), 2, u’’)]; expected_output = [u’error’, (80173.92249999999, u’kelvin**2’)]

test_exponentiation_00008()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), 10, u’degK’)]; expected_output = [u’error’, u’error’]

test_exponentiation_00009()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’kelvin’), 2, u’’)]; expected_output = [(100.0, u’kelvin**2’), (100.0, u’kelvin**2’)]

test_exponentiation_00010()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (2, u’kelvin’)); expected_output = [u’error’, u’error’]]

test_exponentiation_00011()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (500.0, u’millikelvin/kelvin’)); expected_output = [1.4142135623730951, 1.4142135623730951]]

test_exponentiation_00012()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (0.5, u’kelvin/kelvin’)); expected_output = [1.4142135623730951, 1.4142135623730951]]

test_exponentiation_00013()
    test_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u’degC’), (500.0, u’millikelvin/kelvin’)); expected_output = [u’error’, (16.827061537891872, u’kelvin**0.5’)]

Chapter 4. More information
test_inplace_addition_00001()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'kelvin')); expected_output=(110, u'kelvin')]

test_inplace_addition_00002()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'degC')); expected_output=error]

test_inplace_addition_00003()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'degF')); expected_output=error]

test_inplace_addition_00004()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'degR')); expected_output=(105.56, u'kelvin')]

test_inplace_addition_00005()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'delta_degC')); expected_output=(110, u'kelvin')]

test_inplace_addition_00006()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'kelvin'), (10, u'delta_degF')); expected_output=(110, u'kelvin')]

test_inplace_addition_00007()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'kelvin')); expected_output=error]

test_inplace_addition_00008()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'degC')); expected_output=error]

test_inplace_addition_00009()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'degF')); expected_output=error]

test_inplace_addition_00010()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'degR')); expected_output=error]

test_inplace_addition_00011()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'delta_degC')); expected_output=(110, u'degC')]

test_inplace_addition_00012()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degC'), (10, u'delta_degF')); expected_output=(110, u'degC')]

test_inplace_addition_00013()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degF'), (10, u'kelvin')); expected_output=error]

test_inplace_addition_00014()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degF'), (10, u'degC')); expected_output=error]

test_inplace_addition_00015()
  test_inplace_addition(pint.testsuite.test_quantity.TestOffsetUnitMath) [with input=((100, u'degF'), (10, u'degF')); expected_output=error]
test_inplace_addition_00016
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’), (10, u‘degR’)); expected_output = error]

test_inplace_addition_00017
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’), (10, u’delta_degC‘)); expected_output = (110, u‘degF’)]

test_inplace_addition_00018
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’), (10, u’delta_degF’)); expected_output = (118, u‘degF’)]

test_inplace_addition_00019
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u‘kelvin’)); expected_output = (118, u‘degR’)]

test_inplace_addition_00020
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u‘degC’)); expected_output = error]

test_inplace_addition_00021
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u‘degF’)); expected_output = error]

test_inplace_addition_00022
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u‘degR’)); expected_output = (110, u‘degR’)]

test_inplace_addition_00023
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u’delta_degC’)); expected_output = (118, u‘degR’)]

test_inplace_addition_00024
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degR’), (10, u’delta_degF’)); expected_output = (110, u‘degR’)]

test_inplace_addition_00025
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘kelvin’)); expected_output = (110, u‘kelvin’)]

test_inplace_addition_00026
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘degC’)); expected_output = (110, u‘degC’)]

test_inplace_addition_00027
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘degF’)); expected_output = (190, u‘degF’)]

test_inplace_addition_00028
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘degR’)); expected_output = (190, u‘degR’)]

test_inplace_addition_00029
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘delta_degC’)); expected_output = (110, u‘delta_degC’)]

test_inplace_addition_00030
  test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘delta_degC’), (10, u‘delta_degF’)); expected_output = (105.56, u‘delta_degC’)]
test_inplace_addition_00031()

test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, 
  u'delta_degF'), (10, u'kelvin')); expected_output = (65.56, u'kelvin')]

test_inplace_addition_00032()

test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, 
  u'delta_degF'), (10, u'degC')); expected_output = (65.56, u'degC')]

test_inplace_addition_00033()

test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, 
  u'delta_degF'), (10, u'degR')); expected_output = (110, u'degR')]

test_inplace_addition_00034()

test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, 
  u'delta_degF'), (10, u'delta_degC')); expected_output = (118, u'delta_degF')]

test_inplace_addition_00035()

test_inplace_addition (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, 
  u'delta_degF'), (10, u'delta_degF')); expected_output = (110, u'delta_degF')]

test_inplace_exponentiation_00001()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), 0); expected_output = [(10, u'degC'), (10, u'degC')]]

test_inplace_exponentiation_00002()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), 0.5); expected_output = [u'error', (16.827061537891872, u'kelvin**0.5')]]

test_inplace_exponentiation_00003()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), 0); expected_output = [(1.0, u'''), (1.0, u'')]]

test_inplace_exponentiation_00004()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), -1); expected_output = [u'error', (0.003531696980399082, u'kelvin**-1')]]

test_inplace_exponentiation_00005()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), -2); expected_output = [u'error', (1.2472883561359994e-05, u'kelvin**-2')]]

test_inplace_exponentiation_00006()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((0, u'degC'), -2); expected_output = [u'error', (1.3402863367625568e-05, u'kelvin**-2')]]

test_inplace_exponentiation_00007()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), (2, u''')); expected_output = [u'error', (80173.92249999999, u'kelvin**2')]]

test_inplace_exponentiation_00008()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), (10, u'degK')); expected_output = [u'error', u'error']]

test_inplace_exponentiation_00009()

test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'kelvin'), (2, u''')); expected_output = [(100.0, u'kelvin**2'), (100.0, u'kelvin**2')]]
test_inplace_exponentiation_00010()
    test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (2, u'kelvin')); expected_output = [u'error', u'error']]

test_inplace_exponentiation_00011()
    test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (500.0, u'millikelvin/kelvin')); expected_output = [1.4142135623730951, 1.4142135623730951]]

test_inplace_exponentiation_00012()
    test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = (2, (0.5, u'kelvin/kelvin')); expected_output = [1.4142135623730951, 1.4142135623730951]]

test_inplace_exponentiation_00013()
    test_inplace_exponentiation (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), (500.0, u'millikelvin/kelvin')); expected_output = [u'error', (16.827061537891872, u'kelvin**0.5')]]

test_inplace_multiplication_00001()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'kelvin')); expected_output = (1000, u'kelvin**2')]

test_inplace_multiplication_00002()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = error]

test_inplace_multiplication_00003()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degF')); expected_output = error]

test_inplace_multiplication_00004()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (1000, u'kelvin*degR')]

test_inplace_multiplication_00005()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (1000, u'kelvin*delta_degC')]

test_inplace_multiplication_00006()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degF')); expected_output = (1000, u'kelvin*delta_degF')]

test_inplace_multiplication_00007()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = error]

test_inplace_multiplication_00008()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

test_inplace_multiplication_00009()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = error]

test_inplace_multiplication_00010()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degR')); expected_output = error]

test_inplace_multiplication_00011()
    test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degC')); expected_output = error]
test_inplace_multiplication_00012()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degF')); expected_output = error]

test_inplace_multiplication_00013()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'kelvin')); expected_output = error]

test_inplace_multiplication_00014()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

test_inplace_multiplication_00015()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degC')); expected_output = error]

test_inplace_multiplication_00016()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degR')); expected_output = error]

test_inplace_multiplication_00017()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degC')); expected_output = error]

test_inplace_multiplication_00018()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degF')); expected_output = error]

test_inplace_multiplication_00019()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'kelvin')); expected_output = (1000, u'degR*kelvin')]

test_inplace_multiplication_00020()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degC')); expected_output = error]

test_inplace_multiplication_00021()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degF')); expected_output = error]

test_inplace_multiplication_00022()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected_output = (1000, u'degR**2')]

test_inplace_multiplication_00023()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected_output = (1000, u'degR*delta_degC')]

test_inplace_multiplication_00024()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degF')); expected_output = (1000, u'degR*delta_degF')]

test_inplace_multiplication_00025()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (1000, u'delta_degC*kelvin')]

test_inplace_multiplication_00026()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = error]
test_inplace_multiplication_00027()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degC’), (10, u’degF’)); expected_output = error]

test_inplace_multiplication_00028()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degC’), (10, u’degR’)); expected_output = (1000, u’delta_degC*degR’)]

test_inplace_multiplication_00029()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degC’), (10, u’delta_degC’)); expected_output = (1000, u’delta_degC**2’)]

test_inplace_multiplication_00030()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degC’), (10, u’delta_degF’)); expected_output = (1000, u’delta_degC*delta_degF’)]

test_inplace_multiplication_00031()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’kelvin’)); expected_output = (1000, u’delta_degF*kelvin’)]

test_inplace_multiplication_00032()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’deltaC’)); expected_output = error]

test_inplace_multiplication_00033()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’degF’)); expected_output = error]

test_inplace_multiplication_00034()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’degR’)); expected_output = (1000, u’delta_degF*degR’)]

test_inplace_multiplication_00035()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’delta_degC’)); expected_output = (1000, u’delta_degF*delta_degC’)]

test_inplace_multiplication_00036()
test_inplace_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100,
   u’delta_degF’), (10, u’delta_degF’)); expected_output = (1000, u’delta_degF**2’)]

test_inplace_multiplication_with_autoconvert_00001()
test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with in-
put = ((100, u’kelvin’), (10, u’deltaC’)); expected_output = (28315.0, u’kelvin**2’)]

test_inplace_multiplication_with_autoconvert_00002()
test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with in-
put = ((100, u’kelvin’), (10, u’degF’)); expected_output = (26092.78, u’kelvin**2’)]

test_inplace_multiplication_with_autoconvert_00003()
test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with in-
put = ((100, u’degC’), (10, u’kelvin’)); expected_output = (3731.5, u’kelvin**2’)]

test_inplace_multiplication_with_autoconvert_00004()
test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with in-
put = ((100, u’degC’), (10, u’degC’)); expected_output = (105657.42, u’kelvin**2’)]

test_inplace_multiplication_with_autoconvert_00005()
test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with in-
put = ((100, u’degC’), (10, u’degF’)); expected_output = (97365.2, u’kelvin**2’)]
test_inplace_multiplication_with_autoconvert_00006()
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degC}'}, (10, \text{u'\text{degR}'})}); \text{expected_output} = (3731.5, \text{u'kelvin*degR'})]}

\text{test_inplace_multiplication_with_autoconvert_00007()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degC}'}, (10, \text{u'\text{delta_degC}'})}); \text{expected_output} = (3731.5, \text{u'kelvin*delta_degC'})]}

\text{test_inplace_multiplication_with_autoconvert_00008()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degC}'}, (10, \text{u'\text{delta_degF}'})}); \text{expected_output} = (3731.5, \text{u'kelvin*delta_degF'})]}

\text{test_inplace_multiplication_with_autoconvert_00009()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{kelvin}'})}); \text{expected_output} = (3109.28, \text{u'kelvin**2'})]}

\text{test_inplace_multiplication_with_autoconvert_00010()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{degC}'})}); \text{expected_output} = (88039.2, \text{u'kelvin**2'})]}

\text{test_inplace_multiplication_with_autoconvert_00011()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{degF}'})}); \text{expected_output} = (81129.69, \text{u'kelvin**2'})]}

\text{test_inplace_multiplication_with_autoconvert_00012()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{degC}'})}); \text{expected_output} = (3109.28, \text{u'kelvin*degC'})]}

\text{test_inplace_multiplication_with_autoconvert_00013()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{delta_degC}'})}); \text{expected_output} = (3109.28, \text{u'kelvin*delta_degC'})]}

\text{test_inplace_multiplication_with_autoconvert_00014()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degF}'}, (10, \text{u'\text{delta_degF}'})}); \text{expected_output} = (3109.28, \text{u'kelvin*delta_degF'})]}

\text{test_inplace_multiplication_with_autoconvert_00015()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degR}'}, (10, \text{u'\text{degC}'})}); \text{expected_output} = (28315.0, \text{u'\text{degR*kelvin}'})]}

\text{test_inplace_multiplication_with_autoconvert_00016()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{degR}'}, (10, \text{u'\text{degF}'})}); \text{expected_output} = (26092.78, \text{u'\text{degR*kelvin}'})]}

\text{test_inplace_multiplication_with_autoconvert_00017()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{delta_degC}'}, (10, \text{u'\text{degC}'})}); \text{expected_output} = (28315.0, \text{u'\text{delta_degC*kelvin}'})]}

\text{test_inplace_multiplication_with_autoconvert_00018()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{delta_degC}'}, (10, \text{u'\text{degF}'})}); \text{expected_output} = (26092.78, \text{u'\text{delta_degC*kelvin}'})]}

\text{test_inplace_multiplication_with_autoconvert_00019()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{delta_degF}'}, (10, \text{u'\text{degC}'})}); \text{expected_output} = (28315.0, \text{u'\text{delta_degF*kelvin}'})]}

\text{test_inplace_multiplication_with_autoconvert_00020()}
\[\text{test_inplace_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'\text{delta_degF}'}, (10, \text{u'\text{degF}'})}); \text{expected_output} = (26092.78, \text{u'\text{delta_degF*kelvin}'})]}

4.1. Developer reference
test_inplace_subtraction_00001()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘kelvin’)); expected_output = (90, u‘kelvin’)]

test_inplace_subtraction_00002()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘degC’)); expected_output = (-183.15, u‘kelvin’)]

test_inplace_subtraction_00003()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘degF’)); expected_output = (-160.93, u‘kelvin’)]

test_inplace_subtraction_00004()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘degR’)); expected_output = (94.44, u‘kelvin’)]

test_inplace_subtraction_00005()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘delta_degC’)); expected_output = (90, u‘kelvin’)]

test_inplace_subtraction_00006()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘kelvin’),
(10, u‘delta_degF’)); expected_output = (94.44, u‘kelvin’)]

test_inplace_subtraction_00007()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘kelvin’)); expected_output = (363.15, u‘delta_degC’)]

test_inplace_subtraction_00008()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘degC’)); expected_output = (90, u‘delta_degC’)]

test_inplace_subtraction_00009()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘degF’)); expected_output = (112.22, u‘delta_degC’)]

test_inplace_subtraction_00010()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘degR’)); expected_output = (367.59, u‘delta_degC’)]

test_inplace_subtraction_00011()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘delta_degC’)); expected_output = (90, u‘degC’)]

test_inplace_subtraction_00012()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degC’),
(10, u‘delta_degF’)); expected_output = (94.44, u‘degC’)]

test_inplace_subtraction_00013()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’),
(10, u‘kelvin’)); expected_output = (541.67, u‘delta_degF’)]

test_inplace_subtraction_00014()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’),
(10, u‘degC’)); expected_output = (50, u‘delta_degF’)]

test_inplace_subtraction_00015()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u‘degF’),
(10, u‘degF’)); expected_output = (90, u‘delta_degF’)]
test_inplace_subtraction_00016()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), 
    (10, u'degR')); expected_output = (549.67, u'delta_degF')]

test_inplace_subtraction_00017()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), 
    (10, u'delta_degC')); expected_output = (82, u'degF')]

test_inplace_subtraction_00018()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), 
    (10, u'delta_degF')); expected_output = (90, u'degF')]

test_inplace_subtraction_00019()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'kelvin')); expected_output = (82, u'degR')]

test_inplace_subtraction_00020()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'degC')); expected_output = (-409.67, u'degR')]

test_inplace_subtraction_00021()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'degF')); expected_output = (-369.67, u'degR')]

test_inplace_subtraction_00022()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'degR')); expected_output = (90, u'degR')]

test_inplace_subtraction_00023()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'delta_degC')); expected_output = (90, u'delta_degC')]

test_inplace_subtraction_00024()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), 
    (10, u'delta_degF')); expected_output = (90, u'delta_degR')]

test_inplace_subtraction_00025()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'kelvin')); expected_output = (90, u'delta_degC')]

test_inplace_subtraction_00026()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'degC')); expected_output = (90, u'degC')]

test_inplace_subtraction_00027()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'degF')); expected_output = (170, u'degF')]

test_inplace_subtraction_00028()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'degR')); expected_output = (170, u'degR')]

test_inplace_subtraction_00029()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'delta_degC')); expected_output = (90, u'delta_degC')]

test_inplace_subtraction_00030()  
    test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), 
    (10, u'delta_degF')); expected_output = (94.44, u'delta_degC')]
test_inplace_subtraction_00031()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'kelvin')); expected_output = (45.56, u'kelvin')]

test_inplace_subtraction_00032()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = (45.56, u'degC')]

test_inplace_subtraction_00033()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = (90, u'degF')]

test_inplace_subtraction_00034()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degR')); expected_output = (90, u'degR')]

test_inplace_subtraction_00035()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degC')); expected_output = (82, u'delta_degF')]

test_inplace_subtraction_00036()
test_inplace_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degF')); expected_output = (90, u'delta_degF')]

test_inplace_truedivision_00001()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'kelvin')); expected_output = (10, u'')]

test_inplace_truedivision_00002()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = error]

test_inplace_truedivision_00003()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degF')); expected_output = error]

test_inplace_truedivision_00004()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (10, u'kelvin/degR')]

test_inplace_truedivision_00005()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (10, u'kelvin/delta_degC')]

test_inplace_truedivision_00006()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degF')); expected_output = (10, u'kelvin/delta_degF')]

test_inplace_truedivision_00007()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = error]

test_inplace_truedivision_00008()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

test_inplace_truedivision_00009()
test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = error]
test_inplace_truedivision_00010()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degC}'), (10, u'\text{degR}')); expected_output = error]

test_inplace_truedivision_00011()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degC}'), (10, u'\delta_{\text{degC}}')); expected_output = error]

test_inplace_truedivision_00012()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degC}'), (10, u'\delta_{\text{degF}}')); expected_output = error]

test_inplace_truedivision_00013()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degC}'), (10, u'\text{kelvin}')); expected_output = error]

test_inplace_truedivision_00014()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degF}'), (10, u'\text{degC}')); expected_output = error]

test_inplace_truedivision_00015()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degF}'), (10, u'\text{degF}')); expected_output = error]

test_inplace_truedivision_00016()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degF}'), (10, u'\text{degR}')); expected_output = error]

test_inplace_truedivision_00017()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degF}'), (10, u'\delta_{\text{degC}}')); expected_output = error]

test_inplace_truedivision_00018()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degF}'), (10, u'\delta_{\text{degF}}')); expected_output = error]

test_inplace_truedivision_00019()
  test_inplace_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'\text{degR}'), (10, u'\text{kelvin}')); expected_output = (10, u'\text{degR}/\text{kelvin}')]
test_inplace_truedivision_00025()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (10, u'delta_degC/kelvin')]

test_inplace_truedivision_00026()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = error]

test_inplace_truedivision_00027()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degR')); expected_output = error]

test_inplace_truedivision_00028()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degC')); expected_output = (10, u'')]

test_inplace_truedivision_00029()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degF')); expected_output = (10, u'delta_degC/delta_degF')]

test_inplace_truedivision_00030()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = error]

test_inplace_truedivision_00031()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degF')); expected_output = error]

test_inplace_truedivision_00032()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = (10, u'delta_degF/degC')]

test_inplace_truedivision_00033()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = error]

test_inplace_truedivision_00034()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degR')); expected_output = (10, u'delta_degF/degR')]

test_inplace_truedivision_00035()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degC')); expected_output = (10, u'delta_degF/delta_degC')]

test_inplace_truedivision_00036()
    test_inplace_truedivision  (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degF')); expected_output = (10, u')]
test_multiplication_00004()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (1000, u'kelvin*degR')]

test_multiplication_00005()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (1000, u'kelvin*delta_degC')]

test_multiplication_00006()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degF')); expected_output = (1000, u'kelvin*delta_degF')]

test_multiplication_00007()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = error]

test_multiplication_00008()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

test_multiplication_00009()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = error]

test_multiplication_00010()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degR')); expected_output = error]

test_multiplication_00011()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degC')); expected_output = error]

test_multiplication_00012()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degF')); expected_output = error]

test_multiplication_00013()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'kelvin')); expected_output = error]

test_multiplication_00014()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degC')); expected_output = error]

test_multiplication_00015()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degR')); expected_output = error]

test_multiplication_00016()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degC')); expected_output = error]

test_multiplication_00017()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degF')); expected_output = error]
test_multiplication_00019()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'kelvin')); expected_output = (1000, u'degR*kelvin')]

test_multiplication_00020()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degC')); expected_output = error]

test_multiplication_00021()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degF')); expected_output = error]

test_multiplication_00022()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degR')); expected_output = (1000, u'degR**2')]

test_multiplication_00023()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected_output = (1000, u'degR*delta_degC')]

test_multiplication_00024()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degF')); expected_output = (1000, u'degR*delta_degF')]

test_multiplication_00025()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (1000, u'delta_degC*kelvin')]

test_multiplication_00026()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = error]

test_multiplication_00027()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degF')); expected_output = error]

test_multiplication_00028()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degR')); expected_output = (1000, u'delta_degC*degR')]

test_multiplication_00029()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degC')); expected_output = (1000, u'delta_degC**2')]

test_multiplication_00030()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'delta_degF')); expected_output = (1000, u'delta_degC*delta_degF')]

test_multiplication_00031()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'kelvin')); expected_output = (1000, u'delta_degF*kelvin')]

test_multiplication_00032()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = error]

test_multiplication_00033()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = error]
test_multiplication_00034()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'),
    (10, u'degR')); expected_output = (1000, u'delta_degF*degR')]

test_multiplication_00035()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'),
    (10, u'delta_degC')); expected_output = (1000, u'delta_degF*delta_degC')]

test_multiplication_00036()
    test_multiplication (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'),
    (10, u'delta_degF')); expected_output = (1000, u'delta_degF**2')]

test_multiplication_with_autoconvert_00001()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'kelvin'), (10, u'degC')); expected_output = (28315.0, u'kelvin**2')]

test_multiplication_with_autoconvert_00002()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'kelvin'), (10, u'degF')); expected_output = (26092.78, u'kelvin**2')]

test_multiplication_with_autoconvert_00003()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'kelvin')); expected_output = (3731.5, u'kelvin**2')]

test_multiplication_with_autoconvert_00004()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'degC')); expected_output = (105657.42, u'kelvin**2')]

test_multiplication_with_autoconvert_00005()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'degF')); expected_output = (97365.2, u'kelvin**2')]

test_multiplication_with_autoconvert_00006()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'degR')); expected_output = (3731.5, u'kelvin*degR')]

test_multiplication_with_autoconvert_00007()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'delta_degC')); expected_output = (3731.5, u'kelvin*delta_degC')]

test_multiplication_with_autoconvert_00008()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degC'), (10, u'delta_degF')); expected_output = (3731.5, u'kelvin*delta_degF')]

test_multiplication_with_autoconvert_00009()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degF'), (10, u'kelvin')); expected_output = (3109.28, u'kelvin**2')]

test_multiplication_with_autoconvert_00010()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degF'), (10, u'degC')); expected_output = (88039.2, u'kelvin**2')]

test_multiplication_with_autoconvert_00011()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degF'), (10, u'degF')); expected_output = (81129.69, u'kelvin**2')]

test_multiplication_with_autoconvert_00012()
    test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input =
    ((100, u'degF'), (10, u'degR')); expected_output = (3109.28, u'kelvin*degR')]
test_multiplication_with_autoconvert_00013()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degC')); expected_output = (3109.28, u'kelvin*delta_degC')]

test_multiplication_with_autoconvert_00014()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degF')); expected_output = (3109.28, u'kelvin*delta_degF')]

test_multiplication_with_autoconvert_00015()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degC')); expected_output = (28315.0, u'degR*kelvin')]

test_multiplication_with_autoconvert_00016()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degF')); expected_output = (26092.78, u'degR*kelvin')]

test_multiplication_with_autoconvert_00017()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = (28315.0, u'delta_degC*kelvin')]

test_multiplication_with_autoconvert_00018()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degF')); expected_output = (26092.78, u'delta_degC*kelvin')]

test_multiplication_with_autoconvert_00019()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = (28315.0, u'delta_degF*kelvin')]

test_multiplication_with_autoconvert_00020()
  test_multiplication_with_autoconvert (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = (26092.78, u'delta_degF*kelvin')]

test_multiplication_with_scalar_00001()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'kelvin'), 2); expected_output = (20.0, u'kelvin')]

test_multiplication_with_scalar_00002()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'kelvin**2'), 2); expected_output = (20.0, u'kelvin**2')]

test_multiplication_with_scalar_00003()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC'), 2); expected_output = (20.0, u'degC')]

test_multiplication_with_scalar_00004()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'1/degC'), 2); expected_output = error]

test_multiplication_with_scalar_00005()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC**0.5'), 2); expected_output = error]

test_multiplication_with_scalar_00006()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC**2'), 2); expected_output = error]

test_multiplication_with_scalar_00007()
  test_multiplication_with_scalar (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((10, u'degC**-2'), 2); expected_output = error]
test_subtraction_00001()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'kelvin')); expected_output = (90, u'kelvin')]

test_subtraction_00002()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = (-183.15, u'kelvin')]

test_subtraction_00003()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degF')); expected_output = (-160.93, u'kelvin')]

test_subtraction_00004()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (94.44, u'kelvin')]

test_subtraction_00005()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (90, u'kelvin')]

test_subtraction_00006()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degF')); expected_output = (94.44, u'kelvin')]

test_subtraction_00007()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = (363.15, u'delta_degC')]

test_subtraction_00008()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = (90, u'delta_degC')]

test_subtraction_00009()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = (112.22, u'delta_degC')]

test_subtraction_00010()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degR')); expected_output = (367.59, u'delta_degC')]

test_subtraction_00011()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degC')); expected_output = (90, u'degC')]

test_subtraction_00012()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degF')); expected_output = (94.44, u'degC')]

test_subtraction_00013()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'kelvin')); expected_output = (541.67, u'delta_degF')]

test_subtraction_00014()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degC')); expected_output = (50, u'delta_degF')]

test_subtraction_00015()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degF')); expected_output = (90, u'delta_degF')]
test_subtraction_00016()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degF'}), (10, \text{u'degR'})]; expected_output = (549.67, \text{u'delta_degF'})]

test_subtraction_00017()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degF'}), (10, \text{u'delta_degC'})]; expected_output = (82, \text{u'degF'})]

test_subtraction_00018()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degF'}), (10, \text{u'delta_degF'})]; expected_output = (90, \text{u'degF'})]

test_subtraction_00019()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'kelvin'})]; expected_output = (82, \text{u'degR'})]

test_subtraction_00020()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'degC'})]; expected_output = (-409.67, \text{u'degR'})]

test_subtraction_00021()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'degF'})]; expected_output = (-369.67, \text{u'degR'})]

test_subtraction_00022()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'degR'})]; expected_output = (90, \text{u'degR'})]

test_subtraction_00023()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'delta_degC'})]; expected_output = (82, \text{u'degR'})]

test_subtraction_00024()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'degR'}), (10, \text{u'delta_degF'})]; expected_output = (90, \text{u'degR'})]

test_subtraction_00025()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'kelvin'})]; expected_output = (90, \text{u'delta_degC'})]

test_subtraction_00026()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'degC'})]; expected_output = (90, \text{u'delta_degC'})]

test_subtraction_00027()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'degF'})]; expected_output = (170, \text{u'delta_degF'})]

test_subtraction_00028()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'degR'})]; expected_output = (170, \text{u'degR'})]

test_subtraction_00029()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'delta_degC'})]; expected_output = (90, \text{u'delta_degC'})]

test_subtraction_00030()
    test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, \text{u'delta_degC'})], (10, \text{u'delta_degF'})]; expected_output = (94.44, \text{u'delta_degC'})]

88 Chapter 4. More information
test_subtraction_00031()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'kelvin')); expected_output = (45.56, u'kelvin')]

test_subtraction_00032()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degC')); expected_output = (45.56, u'degC')]

test_subtraction_00033()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degF')); expected_output = (90, u'degF')]

test_subtraction_00034()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'degR')); expected_output = (90, u'degR')]

test_subtraction_00035()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degC')); expected_output = (82, u'delta_degF')]

test_subtraction_00036()
test_subtraction (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degF'), (10, u'delta_degF')); expected_output = (90, u'delta_degF')]

test_truedivision_00001()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'kelvin')); expected_output = (10, u'')]

test_truedivision_00002()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degC')); expected_output = error]

test_truedivision_00003()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degF')); expected_output = error]

test_truedivision_00004()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'degR')); expected_output = (10, u'kelvin/degR')]

test_truedivision_00005()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degC')); expected_output = (10, u'kelvin/delta_degC')]

test_truedivision_00006()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'kelvin'), (10, u'delta_degF')); expected_output = (10, u'kelvin/delta_degF')]

test_truedivision_00007()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'kelvin')); expected_output = error]

test_truedivision_00008()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected_output = error]

test_truedivision_00009()
test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected_output = error]
test_truedivision_00010
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degR')); expected output = error]

test_truedivision_00011
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'delta_degC')); expected output = error]

test_truedivision_00012
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degF')); expected output = error]

test_truedivision_00013
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degC'), (10, u'degC')); expected output = error]

test_truedivision_00014
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degC')); expected output = error]

test_truedivision_00015
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degF')); expected output = error]

test_truedivision_00016
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'degR')); expected output = error]

test_truedivision_00017
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degC')); expected output = error]

test_truedivision_00018
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degF'), (10, u'delta_degF')); expected output = error]

test_truedivision_00019
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'kelvin')); expected output = (10, u'degR/kelvin')]

test_truedivision_00020
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degC')); expected output = error]

test_truedivision_00021
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degF')); expected output = error]

test_truedivision_00022
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'degR')); expected output = (10, u'' )]

test_truedivision_00023
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degC')); expected output = (10, u'degR/delta_degC')]

test_truedivision_00024
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'degR'), (10, u'delta_degF')); expected output = (10, u'degR/delta_degF')]

Chapter 4. More information
test_truedivision_00025()
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'kelvin')); expected_output = (10, u'delta_degC/kelvin')]

test_truedivision_00026()
    test_truedivision (pint.testsuite.test_quantity.TestOffsetUnitMath) [with input = ((100, u'delta_degC'), (10, u'degC')); expected_output = error]

class pint.testsuite.test_quantity.TestQuantity (methodName='runTest')

class pint.testsuite.test_quantity.TestQuantityBasicMath (methodName='runTest')

class pint.testsuite.test_quantity.TestQuantityToCompact (methodName='runTest')

class pint.testsuite.test_quantity.TestQuantityWithDefaultRegistry (methodName='runTest')

    classmethod setUpClass()
        Hook method for setting up class fixture before running tests in the class.

class pint.testsuite.test_quantity.TestTimedelta (methodName='runTest')

class pint.testsuite.test_systems.TestGroup (methodName='runTest')

4.1. Developer reference 91
class pint.testsuite.test_systems.TestSystem (methodName='runTest')

class pint.testsuite.test_umath.TestComparisonUfuncs (methodName='runTest')
  Universal functions (ufunc) > Comparison functions

  http://docs.scipy.org/doc/numpy/reference/ufuncs.html#comparison-functions

  greater(x1, x2[, out]) Return the truth value of (x1 > x2) element-wise. greater_equal(x1, x2[, out]) Return
  the truth value of (x1 >= x2) element-wise. less(x1, x2[, out]) Return the truth value of (x1 < x2) element-wise.
  less_equal(x1, x2[, out]) Return the truth value of (x1 <= x2) element-wise. not_equal(x1, x2[, out]) Return (x1
  != x2) element-wise. equal(x1, x2[, out]) Return (x1 == x2) element-wise.

class pint.testsuite.test_umath.TestFloatingUfuncs (methodName='runTest')
  Universal functions (ufunc) > Floating functions

  http://docs.scipy.org/doc/numpy/reference/ufuncs.html#floating-functions

  isreal(x) Returns a bool array, where True if input element is real. iscomplex(x) Returns a bool array, where
  True if input element is complex. isnan(x[, out]) Test element-wise for Not a Number (NaN). isinf(x[, out]) Test
  element-wise for positive or negative infinity. isnan(x[, out]) Test element-wise for Not a Number (NaN), return
  result as a bool array. signbit(x[, out]) Returns element-wise True where signbit is set (less than zero).
  copysign(x1, x2[, out]) Change the sign of x1 to that of x2, element-wise. nextafter(x1, x2[, out]) Return the
  next representable floating-point value after x1 in the direction of x2 element-wise. modf(x[, out1, out2]) Return
  the fractional and integral parts of an array, element-wise. ldexp(x[, out]) Compute y = x1 * 2**x2. fmod(x1, x2[, out])
  Split the number, x, into a normalized fraction (y1) and exponent (y2) fmod(x1, x2[, out]) Return the element-wise remainder of
  division. floor(x[, out]) Return the floor of the input, element-wise. ceil(x[, out]) Return the ceiling of the input, element-wise.
  trunc(x[, out]) Return the truncated value of the input, element-wise.

class pint.testsuite.test_umath.TestMathUfuncs (methodName='runTest')
  Universal functions (ufunc) > Math operations

  http://docs.scipy.org/doc/numpy/reference/ufuncs.html#math-operations

  add(x1, x2[, out]) Add arguments element-wise. subtract(x1, x2[, out]) Subtract arguments, element-wise.
  multiply(x1, x2[, out]) Multiply arguments element-wise. divide(x1, x2[, out]) Divide arguments element-wise.
  logaddexp(x1, x2[, out]) Logarithm of the sum of exponentiations of the inputs. logaddexp2(x1, x2[, out])
  Logarithm of the sum of exponentiations of the inputs in base-2. true_divide(x1, x2[, out]) Returns a true
  division of the inputs, element-wise. floor_divide(x1, x2[, out]) Return the largest integer smaller or equal
  to the division of the inputs. negative(x[, out]) Returns an array with the negative of each element of the
  original array. power(x1, x2[, out]) First array elements raised to powers from second array, element-wise.
  NOT IMPLEMENTED remainder(x1, x2[, out]) Return element-wise remainder of division. mod(x1, x2[, out])
  Return element-wise remainder of division. absolute(x[, out]) Calculate the absolute value element-wise. rint(x[, out])
  Round elements of the array to the nearest integer. sign(x[, out]) Returns an element-wise indication of the sign of a number.
  conj(x[, out]) Return the complex conjugate, element-wise. exp(x[, out]) Calculate the exponential of all elements in the input array.
  exp2(x[, out]) Calculate 2**p for all p in the input array. log(x[, out]) Natural logarithm, element-wise. log2(x[, out])
  Base-2 logarithm of x. log10(x[, out]) Return the base 10 logarithm of the input array, element-wise. expm1(x[, out])
  Calculate exp(x) - 1 for all elements in the array. log1p(x[, out]) Return the natural logarithm of one plus the input array, element-wise.
  sqrt(x[, out]) Return the positive square-root of an array, element-wise. square(x[, out]) Return the element-wise square of the input.
  reciprocal(x[, out]) Return the reciprocal of the argument, element-wise. ones_like(x[, out]) Returns an array of ones with the same shape and type as a given
  array.

class pint.testsuite.test_umath.TestTrigUfuncs (methodName='runTest')
  Universal functions (ufunc) > Trigonometric functions

  http://docs.scipy.org/doc/numpy/reference/ufuncs.html#trigonometric-functions
sin(x[, out]) Trigonometric sine, element-wise. cos(x[, out]) Cosine elementwise. tan(x[, out]) Compute tangent element-wise. arcsin(x[, out]) Inverse sine, element-wise. arccos(x[, out]) Trigonometric inverse cosine, element-wise. arctan(x[, out]) Trigonometric inverse tangent, element-wise. arctan2(x1, x2[, out]) Element-wise arc tangent of x1/x2 choosing the quadrant correctly. hypot(x1, x2[, out]) Given the “legs” of a right triangle, return its hypotenuse. sinh(x[, out]) Hyperbolic sine, element-wise. cosh(x[, out]) Hyperbolic cosine, element-wise. tanh(x[, out]) Compute hyperbolic tangent element-wise. arcsinh(x[, out]) Inverse hyperbolic sine elementwise. arccosh(x[, out]) Inverse hyperbolic cosine, elementwise. arctanh(x[, out]) Inverse hyperbolic tangent elementwise. deg2rad(x[, out]) Convert angles from degrees to radians. rad2deg(x[, out]) Convert angles from radians to degrees.

class pint.testsuite.test_umath.TestUFuncs (methodName='runTest')

class pint.testsuite.test_unit.TestCompatibleUnits (methodName='runTest')

setUp()

Hook method for setting up the test fixture before exercising it.

class pint.testsuite.test_unit.TestConvertWithOffset (methodName='runTest')

test_to_and_from_offset_units_00001()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'degC': 1}, {'degC': 1}); expected_output = 10]

test_to_and_from_offset_units_00002()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'degC': 1}, {'kelvin': 1}); expected_output = 283.15]

test_to_and_from_offset_units_00003()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'degC': 1}, {'millimeter': 1, 'degC': 1, 'meter': -1}); expected_output = error]

test_to_and_from_offset_units_00004()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'degC': 1}, {'millimeter': 1, 'kelvin': 1, 'meter': -1}); expected_output = 283150]

test_to_and_from_offset_units_00005()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'kelvin': 1}, {'degC': 1}); expected_output = -263.15]

test_to_and_from_offset_units_00006()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'kelvin': 1}, {'kelvin': 1}); expected_output = 10]

test_to_and_from_offset_units_00007()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'kelvin': 1}, {'millimeter': 1, 'degC': 1, 'meter': -1}); expected_output = error]

test_to_and_from_offset_units_00008()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ({'kelvin': 1}, {'millimeter': 1, 'kelvin': 1, 'meter': -1}); expected_output = 10000]

test_to_and_from_offset_units_00009()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ([u’millimeter’: 1, u’degC’: 1, u’meter’: -1], [u’degC’: 1]); expected_output = error]

test_to_and_from_offset_units_00010()
test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input = ([u’millimeter’: 1, u’degC’: 1, u’meter’: -1], [u’kelvin’: 1]); expected_output = error]

4.1. Developer reference
test_to_and_from_offset_units_00011()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’degC’: 1, u’meter’: -1}, {u’millimeter’: 1, u’degC’: 1, u’meter’: -1});
    expected_output = 10]

test_to_and_from_offset_units_00012()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’degC’: 1, u’meter’: -1}, {u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1});
    expected_output = error]

test_to_and_from_offset_units_00013()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1}, {u’degC’: 1}); expected_output = -273.14]

test_to_and_from_offset_units_00014()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1}, {u’kelvin’: 1}); expected_output = 0.01]

test_to_and_from_offset_units_00015()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1}, {u’millimeter’: 1, u’degC’: 1, u’meter’: -1});
    expected_output = error]

test_to_and_from_offset_units_00016()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1}, {u’millimeter’: 1, u’kelvin’: 1, u’meter’: -1});
    expected_output = 10]

test_to_and_from_offset_units_00017()
    test_to_and_from_offset_units (pint.testsuite.test_unit.TestConvertWithOffset) [with input =
    ({u’degC’: 2}, {u’kelvin’: 2}); expected_output = error]

class pint.testsuite.test_unit.TestRegistry (methodName='runTest')

class pint.testsuite.test_util.TestRegistryWithDefaultRegistry (methodName='runTest')

    classmethod setUpClass()
        Hook method for setting up class fixture before running tests in the class.

class pint.testsuite.test_unit.TestUnit (methodName='runTest')

class pint.testsuite.test_util.TestGraph (methodName='runTest')

class pint.testsuite.test_util.TestMatrix (methodName='runTest')

class pint.testsuite.test_util.TestOtherUtils (methodName='runTest')

class pint.testsuite.test_util.TestParseHelper (methodName='runTest')

class pint.testsuite.test_util.TestStringHelper (methodName='runTest')

class pint.testsuite.test_util.TestToolsContainer (methodName='runTest')

class pint.testsuite.test_util.TestToolsContainer (methodName='runTest')
4.2 Contributing to Pint

You can contribute in different ways:

4.2.1 Report issues

You can report any issues with the package, the documentation to the Pint issue tracker. Also feel free to submit feature requests, comments or questions.

4.2.2 Contribute code

To contribute fixes, code or documentation to Pint, fork Pint in github and submit the changes using a pull request against the master branch.

- If you are fixing a bug, add a test to test_issues.py Also add “Close #<bug number> as described in the github docs.
- If you are submitting new code, add tests and documentation.

Pint uses bors-ng as a merge bot and therefore every PR is tested before merging.
In any case, feel free to use the issue tracker to discuss ideas for new features or improvements.

4.3 Frequently asked questions

4.3.1 Why the name Pint?

Pint is a unit and sounds like Python in the first syllable. Most important, it is a good unit for beer.

4.3.2 You mention other similar Python libraries. Can you point me to those?

natu
Buckingham
Magnitude
SciMath
Python-quantities
Unum
Units
udunitspy
SymPy
cf units
astropy units
yt
measurement
If you're aware of another one, please contribute a patch to the docs.
The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, “Small Forces,” used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). The output from the SM_FORCES application code as required by a MSOP Project Software Interface Specification (SIS) was to be in metric units of Newton-seconds (N-s). Instead, the data was reported in English units of pound-seconds (lbf-s). The Angular Momentum Desaturation (AMD) file contained the output data from the SM_FORCES software. The SIS, which was not followed, defines both the format and units of the AMD file generated by ground-based computers. Subsequent processing of the data from AMD file by the navigation software algorithm therefore, underestimated the effect on the spacecraft trajectory by a factor of 4.45, which is the required conversion factor from force in pounds to Newtons. An erroneous trajectory was computed using this incorrect data.
Python Module Index

p

pint, 41
pint.babel_names, 46
pint.compat.chainmap, 63
pint.compat.lru_cache, 64
pint.compat.meta, 64
pint.compat.tokenize, 64
pint.context, 46
pint.converters, 47
pint.definitions, 47
pint.errors, 48
pint.formatting, 48
pint.matplotlib, 49
pint.measurement, 50
pint.pint_eval, 50
pint.quantity, 50
pint.registry, 52
pint.registry_helpers, 58
pint.systems, 59
pint.testsuite.helpers, 65
pint.testsuite.parameterized, 65
pint.testsuite.test_babel, 65
pint.testsuite.test_contexts, 65
pint.testsuite.test_converters, 65
pint.testsuite.test_definitions, 65
pint.testsuite.test_errors, 65
pint.testsuite.test_formatter, 65
pint.testsuite.test_infer_base_unit, 65
pint.testsuite.test_issues, 66
pint.testsuite.test_measurement, 66
pint.testsuite.test_numpy, 66
pint.testsuite.test_pint_eval, 66
pint.testsuite.test_pitheorem, 66
pint.testsuite.test_quantity, 66
pint.testsuite.test_systems, 91
pint.testsuite.test_umath, 92
pint.testsuite.test_unit, 93
pint.testsuite.test_util, 94
pint.unit, 60
pint.util, 61
test_addition_00003()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00004()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00005()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00006()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00007()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00008()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00009()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00010()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00011()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00012()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00013()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00014()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00015()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 67

test_addition_00016()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00017()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00018()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00019()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00020()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00021()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00022()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00023()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00024()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00025()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00026()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00027()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00028()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00029()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68

test_addition_00030()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00031()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00032()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00033()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00034()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00035()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_addition_00036()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 69

test_cross()
  (pint.testsuite.test_numpy.TestNumpyNeedsSubclassing
   method), 66

test_diff()
  (pint.testsuite.test_numpy.TestNumpyNeedsSubclassing
   method), 66

test_division_with_scalar_00001()
  (pint.testsuite.test_quantity.TestOffsetUnitMath
   method), 68
null
test_subtraction_00023()
(pint.testsuite.test_quantity.TestOffsetUnitMath method), 88

(test_to_and_from_offset_units_00005()
(pint.testsuite.test_unit.TestConvertWithOffset method), 93

(test_to_and_from_offset_units_00006()
(pint.testsuite.test_unit.TestConvertWithOffset method), 93

(test_to_and_from_offset_units_00007()
(pint.testsuite.test_unit.TestConvertWithOffset method), 93

(test_to_and_from_offset_units_00008()
(pint.testsuite.test_unit.TestConvertWithOffset method), 93

(test_to_and_from_offset_units_00009()
(pint.testsuite.test_unit.TestConvertWithOffset method), 93

(test_to_and_from_offset_units_00010()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00011()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00012()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00013()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00014()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00015()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00016()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00017()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00018()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_to_and_from_offset_units_00019()
(pint.testsuite.test_unit.TestConvertWithOffset method), 94

(test_trapz() (pint.testsuite.test_numpy.TestNumpyNeedsSubclassing method), 66

(test_truedivision_00001()
(pint.testsuite.test_quantity.TestOffsetUnitMath method), 89

(test_truedivision_00002() (pint.testsuite.test_quantity.TestOffsetUnitMath method), 93

Index 113
method), 89

test_truedivision_00003()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00004()
(pint.testsuite.test Quantity.TestOffsetUnitMath
method), 89

test_truedivision_00005()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00006()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00007()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00008()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00009()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00010()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 89

test_truedivision_00011()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00012()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00013()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00014()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00015()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00016()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00017()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00018()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00019()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00020()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00021()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00022()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00023()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00024()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00025()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 90

test_truedivision_00026()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00027()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00028()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00029()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00030()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00031()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00032()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00033()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00034()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00035()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_truedivision_00036()
(pint.testsuite.test_quantity.TestOffsetUnitMath
method), 91

test_unwrap()
(pint.testsuite.test_numpy.TestNumpyNeedsSubclassing
method), 66

TestBabel
class in pint.testsuite.test_babel, 65

TestCompareZero
class in pint.testsuite.test_quantity, 66
UnitRegistry (class in pint), 44
UnitRegistry (class in pint.registry), 57
units (pint.Quantity attribute), 44
units (pint.quantity.Quantity attribute), 52
UnitsContainer (class in pint.util), 62
UnitStrippedWarning, 46, 48
untokenize() (in module pint.compat.tokenize), 65

W
with_context() (pint.registry.ContextRegistry method), 55
with_metaclass() (in module pint.compat.meta), 64
wraps() (in module pint.registry_helpers), 58
wraps() (pint.registry.UnitRegistry method), 57
wraps() (pint.UnitRegistry method), 45