# pint Documentation

***Release 0.9.dev0***

**Hernan E. Grecco**

**May 26, 2017**

# Contents

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 2.6+ and 3.2+ with no other dependency. It is licensed under BSD.

It is extremely easy and natural to use:

```
>>> 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:

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

CHAPTER 1

# Quick Installation

To install Pint, simply:

```
$ pip install pint
```

or utilizing conda, with the conda-forge channel:

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

and then simply enjoy it!

CHAPTER 2

# Design principles

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 $3.14 \pm 0.01$) 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 3.0+.

CHAPTER 3

# User Guide

## Installation

Pint has no dependencies except Python itself. In runs on Python 2.6 and 3.0+.

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()
```

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.

### Getting the code

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

```
$ git clone git://github.com/hgrecco/pint.git
```

Download the tarball:

```
$ curl -OL https://github.com/hgrecco/pint/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/hgrecco/pint/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

# Tutorial

## Converting Quantities

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

```
>>> 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:

```
>>> 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:

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

and can handle mathematical operations between:

```
>>> 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:

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

This method returns a new object leaving the original intact as can be seen by:

```
>>> 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:

```
>>> 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:

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

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

```
>>> 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
```

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

```
>>> 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:

```
>>> 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:

```
>>> speed = 23 * ureg.snail_speed
Traceback (most recent call last):
```

```
...
pint.errors.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*

## String parsing

Pint can also handle units provided as strings:

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

or using the registry as a callable for a short form for *parse_expression*:

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

or using the *Quantity* constructor:

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

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

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

or:

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

or leave out the * altogether:

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

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

```
>>> 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:

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

or

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

**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.

## 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
```

**Note:** In Python 2.6, unnumbered placeholders are invalid. Therefore you need to write *{0}* instead of *{}*, *{0!s}* instead of *{!s}* in string formatting operations.

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

```
>>> accel = 1.3 * ureg['meter/second**2']
>>> # Pretty print
>>> 'The pretty representation is {:P}'.format(accel)
'The pretty representation is 1.3 meter/second²'
>>> # Latex print
>>> 'The latex representation is {:L}'.format(accel)
'The latex representation is 1.3\\ \\frac{\\mathrm{meter}}{\\mathrm{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 ~:

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

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

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'
```

## 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*:

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

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

```
from . import ureg, Q_

length = 10 * ureg.meter
my_speed = Q_(20, 'm/s')
```

If you are pickling and unplicking 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:

```
>>> 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
```

## 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:

```
>>> import numpy as np
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
>>> Q_ = ureg.Quantity
```

and then we create a quantity the standard way

```
>>> 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:

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

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

```
>>> print(legs1.to('kilometer'))
[ 0.003  0.004] kilometer
>>> print(legs1.dimensionality)
[length]
>>> legs1.to('joule')
Traceback (most recent call last):
...
pint.errors.DimensionalityError: Cannot convert from 'meter' ([length]) to 'joule'␣
↪([length] ** 2 * [mass] / [time] ** 2)
```

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

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

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

```
>>> 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:

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

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

```
>>> 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:

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

## 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, isinf, isnan, signbit, copysign, nextafter, modf, ldexp, frexp, fmod, floor, ceil, trunc

And the following ndarrays methods and functions:

- 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

## 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 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.

## Temperature conversion

Unlike meters and seconds, the temperature units fahrenheits and celsius are non-multiplicative units. These temperature 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:

```
>>> 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:

```
>>> 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).

```
>>> increase = 12.3 * ureg.delta_degC
>>> print(increase.to(ureg.kelvin))
12.3 kelvin
>>> print(increase.to(ureg.delta_degF))
22.14 delta_degF
```

Subtraction of two temperatures given in offset units yields a *delta* unit:

```
>>> Q_(25.4, ureg.degC) - Q_(10., ureg.degC)
<Quantity(15.4, 'delta_degC')>
```

You can add or subtract a quantity with *delta* unit and a quantity with offset unit:

```
>>> 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')>
```

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

```
>>> heating_rate = 0.5 * ureg.kelvin/ureg.min
>>> Q_(10., ureg.degC) + heating_rate * Q_(30, ureg.min)
Traceback (most recent call last):
        ...
pint.errors.OffsetUnitCalculusError: Ambiguous operation with offset unit (degC,␣
↪kelvin).
```

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

```
>>> Q_(10., ureg.degC).to(ureg.kelvin) + heating_rate * Q_(30, ureg.min)
<Quantity(298.15, 'kelvin')>
```

or convert the absolute unit to a *delta* unit:

```
>>> Q_(10., ureg.degC) + heating_rate.to('delta_degC/min') * Q_(30, ureg.min)
<Quantity(25.0, 'degC')>
```

In contrast to subtraction, the addition of quantities with offset units is ambiguous, e.g. for *10 degC + 100 degC* two different result are reasonable depending on the context, *110 degC* or *383.15 °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 *delta* units are multiplicative:

```
>>> speed = 60. * ureg.delta_degC / ureg.min
>>> print(speed.to('delta_degC/second'))
1.0 delta_degC / second
```

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:

```
>>> ureg = UnitRegistry()
>>> home = 25.4 * ureg.degC
Traceback (most recent call last):
    ...
pint.errors.OffsetUnitCalculusError: Ambiguous operation with offset unit (degC).
>>> Q_(25.4, ureg.degC)
<Quantity(25.4, 'degC')>
```

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.

```
>>> 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.

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

You can change the behaviour at any time:

[1] If the exponent is +1, the quantity will not be converted to base unit but remains unchanged.

```
>>> ureg.autoconvert_offset_to_baseunit = False
>>> 1/T
Traceback (most recent call last):
    ...
pint.errors.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:

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

but not here:

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

You can override this behaviour:

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

Note that the magnitude is left unchanged:

```
>>> 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:

```
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.

## 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:

```
>>> 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:

```
>>> 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:

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

Or in the decorator format:

```
>>> @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)

## Strict Mode

By default, the function is wrapped in *strict* mode. In this mode, the input arguments assigned to units must be a Quantities.

```
>>> mypp(1. * ureg.meter)
<Quantity(2.0064092925890407, 'second')>
>>> mypp(1.)
Traceback (most recent call last):
...
ValueError: A wrapped function using strict=True requires quantity for all arguments␣
↪with not None units. (error found for meter, 1.0)
```

To enable using non-Quantity numerical values, set strict to False‘.

```
>>> mypp_ns = ureg.wraps(ureg.second, ureg.meter, False)(pendulum_period)
>>> mypp_ns(1. * ureg.meter)
<Quantity(2.0064092925890407, 'second')>
>>> mypp_ns(1.)
<Quantity(2.0064092925890407, 'second')>
```

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

## 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
```

## Specifying relations between arguments

In certain cases the actual units but just their relation. This is done using string starting with the equal sign =:

```
>>> @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:

```
>>> @ureg.wraps('=A**2*B', ('=A', '=A*B', '=B'))
... def some_function(x, y, z):
...     pass
```

## Ignoring an argument or return value

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

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

## 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.

```
>>> mypp = ureg.check('[length]')(pendulum_period)
```

In the decorator format:

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

## 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:

```
>>> 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:

```
>>> 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 **DEFAULT_REGISTRY** is used for unpickling and this might be different from the one that was used during pickling. If you want to have control over the deserialization, the best way is to create a tuple with the magnitude and the units:

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

And then you can just pickle that:

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

To unpickle, just

```
>>> 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:

```
>>> 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')
```

(Check out the serialize docs for more information)

## Buckingham Pi Theorem

Buckingham $\pi$ 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 *[length]/[time]*, *[time]* and *[length]* respectively.

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> ureg.pi_theorem({'V': 'meter/second', 'T': 'second', 'L': 'meter'})
[{'V': 1.0, 'T': 1.0, 'L': -1.0}]
```

or quantities:

```
>>> 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}]
```

## 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.

```
>>> 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)

## 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.

```
>>> 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

## 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 your do this directly:

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

You probably want to use the relation *frequency = speed_of_light / wavelength*:

```
>>> (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.

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

or with the abbreviated form:

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

Contexts can be also enabled for blocks of code using the *with* statement:

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

If you need a particular context in all your code, you can enable it for all operations with the registry:

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

To disable the context, just call:

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

### Enabling multiple contexts

You can enable multiple contexts:

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

This works also using the *with* statement:

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

or in the registry:

```
>>> 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:

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

## 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.

```
>>> 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.

```
>>> substance = 95 * ureg('g')
>>> substance.to('moles', 'chemistry', mw = 5 * ureg('g/mol'))
<Quantity(19.0, 'mole')>
```

## 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.

```
>>> wl = 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'_**

### 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.

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

## 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()
>>> book_length = (20. * ureg.centimeter).plus_minus(2.)
>>> 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.

# Defining units

## 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*. The first word is always the canonical name. Next comes the definition (based on other units). Finally, a list of aliases, separated by equal signs.

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:

```
>>> from pint import UnitRegistry
>>> # First we create the registry.
>>> ureg = UnitRegistry()
>>> # Then we append the new definitions
>>> ureg.load_definitions('/your/path/to/my_def.txt')
```

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:

```
>>> 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.

## Programmatically

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

```
>>> 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.

You can also add prefixes programmatically:

```
>>> ureg.define('myprefix- = 30 = my-')
```

where the number indicates the multiplication factor.

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

# 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.

## Use magnitudes when possible

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

```
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:

```
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.

### 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)
10000 loops, best of 3: 65.4 µs per loop
```

## Different Unit Systems (and default units)

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

```
>>> import pint
>>> ureg = pint.UnitRegistry(system='mks')
>>> ureg.default_system
'mks'
```

This has an effect in the base units. For example:

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

But if you change to cgs:

```
>>> ureg.default_system = 'cgs'
>>> q.to_base_units()
<Quantity(100.0, 'centimeter / second')>
```

or more drastically to:

```
>>> 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:

```
>>> ureg.default_system = 'mks'
>>> ureg.get_compatible_units('meter')
frozenset({<Unit('light_year')>, <Unit('angstrom')>})
```

or for imperial units:

```
>>> 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:

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

Or which units are available within a particular system:

```
>>> 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:

```
>>> (1 * ureg.sys.imperial.pint).to('liter')
<Quantity(0.5682612500000002, 'liter')>
>>> (1 * ureg.sys.US.pint).to('liter')
<Quantity(0.47317647300000004, 'liter')>
>>> (1 * ureg.sys.US.pint).to(ureg.sys.imperial.pint)
<Quantity(0.8326741846289889, 'imperial_pint')>
```

CHAPTER 4

# More information

## Contributing to Pint

You can contribute in different ways:

### 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.

### 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.

## Frequently asked questions

### 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.

### You mention other similar Python libraries. Can you point me to those?

natu

Buckingham

Magnitude

SciMath

Python-quantities

Unum

Units

udunitspy

SymPy

Units

cf units

If your are aware of another one, please contribute a patch to the docs.

# CHAPTER 5

## One last thing

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 Newtonseconds (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.

*Mars Climate Orbiter Mishap Investigation Phase I Report* PDF