pint Documentation

Release 0.1

Hernan E. Grecco

June 24, 2014

Contents

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.

It is distributed with a comprehensive list of physical units, prefixes and constants. Due to it's modular design, you can extend (or even rewrite!) the complete list without changing the source code.

It has a complete test coverage. It runs in Python 2.7 and 3.X with no other dependency. It licensed under BSD.

CHAPTER 1

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 programed 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 simple and easy to edit text file. 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 latex and pretty formatting.

Small codebase: small and easy to maintain codebase with a flat hierarchy. It is a single stand-alone module that can be installed as a package or added side by side to your project.

Dependency free: it depends only on Python and it's standard library.

Python 2 and 3: a single codebase that runs unchanged in Python 2.6+ and Python 3.0+.

Experimental advanced NumPy support: While NumPy is not a requirement for Pint, when available ndarray methods and ufuncs can be used in Quantity objects.

1.1 Soon

Handle temperature conversion: it can convert between units with different point of reference, like positions on a map or absolute temperature scales.

CHAPTER 2

Where to start

2.1 Getting Pint

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

You can install it using pip as a package:

```
$ sudo pip install pint
```

or you can also install by downloading the source code and then running:

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

Alternatively, you can just copy two files (*pint.py* and *default_en.txt*) to your project folder. This provides a good alternative to simplify packaging.

2.2 Development

You can follow and contribute to Pint's development using git:

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

Bugs, feature requests, and questions can be directed to the github website.

2.3 Tutorial

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 quantities objects (*Quantity*). Physical quantities can be queried for the magnitude and units:

```
>>> print(distance.magnitude)
24.0
>>> print(distance.units)
meter
```

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(7086.61417323, '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 )
<Quantity(7086.61417323, 'inch / minute')>
>>> print(speed)
7086.61417323 inch / minute
```

If you ask Pint to perform and invalid conversion:

```
>>> speed.to(ureg.joule)
Traceback (most recent call last):
...
pint.pint.DimensionalityError: Cannot convert from 'inch / minute' (length / time) to 'joule' (length
```

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 in-built parse allows to recognize 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.pint.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*

2.3.1 String parsing

Pint can also handle units provided as strings:

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

or via de *Quantity* constructor:

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

Numbers are also parsed:

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

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

```
>>> input = '2.54 * centimeter to inch' # this is obtained from user input
>>> src, dst = input.split(' to ')
>>> Q_(src).to(dst)
<Quantity(1.0, 'inch')>
```

Take a look at *qconvert.py* within the examples folder for a full script.

2.3.2 String formatting

Pint's physical quantities can be easily printed:

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

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

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

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

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

The same is true for repr (*r*), latex (*l*) and pretty (*p*) specs.

2.3.3 Using it in your projects

If you use Pint in multiple modules within you 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
Q_ = UnitRegistry().Quantity
```

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

```
from . import Q_

my_speed = Quantity(20, 'm/s')
```

2.4 Defining units

2.4.1 Programmatically

You can easily add units to the registry programmatically. Let's add a dog_year (sometimes writen 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.add_unit('dog_year', Q_(52, 'day'), ('dy', ))

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

Units added programmatically are forgotten when the UnitRegistry object is deleted.

2.4.2 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.add_from_file('/your/path/to/my_def.txt')
```

If you make a translation of the default units, 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')
```

2.4.3 Prefixes

You can also add prefixes programmatically:

```
>>> ureg.add_prefix('myprefix', 30, 'my')
```

where the number indicates the multiplication factor.

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.

2.5 Frequently asked questions

2.5.1 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.

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

Buckingham

Magnitude

SciMath

Python-quantities

Unum

Units

If your are aware of another one, please let me know.

Note: *A small technical note*

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

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*