quantities Documentation

Release 0.12.0+5.g4a9d268

Darren Dale

Aug 30, 2017

Contents

Quantities is designed to handle arithmetic and conversions of physical quantities, which have a magnitude, dimensionality specified by various units, and possibly an uncertainty. Quantities builds on the popular numpy library and is designed to work with numpy's standard ufuncs, many of which are already supported.

Quantities is actively developed, and while the current features and API are stable, test coverage is incomplete and the package is not ready for production use. Python-2.6.0 or later is required.

CHAPTER 1

User's Guide

Installation

Prerequisites

Quantities has a few dependencies:

- Python (>=2.7)
- NumPy (>=1.8.2)

Source Code Installation

To install Quantities, download the Quantites sourcecode from PyPi and run "python setup.py install" in the quantities source directory, or run "pip install quantities".

Development

You can follow and contribute to Quantities' development using git:

```
git clone git@github.com:python-quantities/python-quantities.git
```

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

Quick-start tutorial

Quantities is designed to handle arithmetic and conversions of physical quantities, which have a magnitude, dimensionality specified by various units, and possibly an uncertainty. Quantities is designed to work with numpy's standard ufuncs, many of which are already supported. The package is actively developed, and while the current features and API are stable, test coverage is incomplete and so the package is not suggested for production use.

It is strongly suggested to import quantities to its own namespace, so units and constants variables are not accidentally overwritten:

```
>>> import quantities as pq
```

Here pq stands for "physical quantities" or "python quantities". There are a number of ways to create a quantity. In practice, it is convenient to think of quantities as a combination of a magnitude and units. These two quantities are equivalent:

```
>>> import numpy as np
>>> q = np.array([1,2,3]) * pq.J
>>> q = [1,2,3] * pq.J
>>> print q
[ 1.  2.  3.] J
```

The Quantity constructor can also be used to create quantities, similar to numpy.array. Units can be designated using a string containing standard unit abbreviations or unit names. For example:

```
>>> q = pq.Quantity([1,2,3], 'J')
>>> q = pq.Quantity([1,2,3], 'joules')
```

Units are also available as variables, and can be passed to Quantity:

```
>>> q = pq.Quantity([1,2,3], pq.J)
```

You can modify a quantity's units in place:

```
>>> q = 1 * pq.m
>>> q.units = pq.ft
>>> print q
3.280839895013123 ft
```

or equivalently:

```
>>> q = 1 * pq.meter
>>> q.units = 'ft' # or 'foot' or 'feet'
>>> print q
3.280839895013123 ft
```

Note that, with strings, units can be designated using plural variants. Plural variants of the module variables are not available at this time, in the interest of keeping the units namespace somewhat manageable. *q.units = 'feet'* will work, *q.units = pq.feet* will not.

The units themselves are special objects that can not be modified in place:

```
>>> pq.meter.units = 'feet'
AttributeError: can not modify protected units
```

Instead of modifying a quantity in place, you can create a new quantity, rescaled to the new units:

```
>>> q = 300 * pq.ft * 600 * pq.ft
>>> q2 = q.rescale('US_survey_acre')
>>> print q2
4.13221487605 US_survey_acre
```

but rescaling will fail if the requested units fails a dimensional analysis:

```
>>> q = 10 * pq.joule
>>> q2 = q.rescale(pq.watt)
ValueError: Unable to convert between units of "J" and "W"
```

Quantities can not be rescaled in place if the unit conversion fails a dimensional analysis:

```
>>> q = 10 * pq.joule
>>> q.units = pq.watts
ValueError: Unable to convert between units of "J" and "W"
>>> print q
10.0 J
```

Quantities will attempt to simplify units when the users intent is unambiguous:

```
>>> q = (10 * pq.meter)**3
>>> q2 = q/(5*pq.meter)**2
>>> print q2
40 m
```

Quantities will not try to guess in an ambiguous situation:

```
>>> q = (10 * pq.meter)**3
>>> q2 = q/(5*pq.ft)**2
>>> print q2
40 m**3/ft**2
```

In that case, it is not clear whether the user wanted ft converted to meters, or meters to feet, or neither. Instead, you can obtain a new copy of the quantity in its irreducible units, which by default are SI units:

```
>>> q = (10 * pq.meter)**3
>>> q2 = q/(5*pq.ft)**2
>>> print q2
40 m**3/ft**2
>>> qs = q2.simplified
>>> print qs
430.556416668 m
```

It is also possible to customize the units in which simplified quantities are expressed:

```
>>> pq.set_default_units('cgs')
>>> print pq.J.simplified
10000000.0 g*cm**2/s**2
>>> pq.set_default_units(length='m', mass='kg')
```

There are times when you may want to treat a group of units as a single compound unit. For example, surface area per unit volume is a fairly common quantity in materials science. If expressed in the usual way, the quantity will be expressed in units that you may not recognize:

```
>>> q = 1 * pq.m**2 / pq.m**3
>>> print q
1.0 1/m
```

Here are some tricks for working with these compound units, which can be preserved:

```
>>> q = 1 * pq.CompoundUnit("m**2/m**3")
>>> print q
1.0 (m**2/m**3)
```

and can be simplified:

```
>>> qs = q.simplified
>>> qs
1.0 1/m
```

and then rescaled back into compound units:

```
>>> q2 = qs.rescale(CompoundUnit("m**2/m**3"))
>>> print q2
1.0 (m**2/m**3)
```

Compound units can be combined with regular units as well:

```
>>> q = 1 * pq.CompoundUnit('parsec/cm**3') * pq.cm**2
>>> print q
1.0 cm**2*(parsec/cm**3)
```

It is easy to define a unit that is not already provided by quantities. For example:

```
>>> uK = pq.UnitQuantity('microkelvin', pq.degK/1e6, symbol='uK')
>>> print uK
1 uK (microkelvin)
>>> q = 1000*uK
>>> print q.simplified
0.001 K
```

There is also support for quantities with uncertainty:

```
>>> q = UncertainQuantity(4,J,.2)
>>> q
4.0*J
+/-0.2*J (1 sigma)
```

By assuming that the uncertainties are uncorrelated, the uncertainty can be propagated during arithmetic operations:

```
>>> length = UncertainQuantity(2.0, m, .001)
>>> width = UncertainQuantity(3.0, m, .001)
>>> area = length*width
>>> area
6.0*m**2
+/-0.00360555127546*m**2 (1 sigma)
```

In that case, the measurements of the length and width were independent, and the two uncertainties presumed to be uncorrelated. Here is a warning though:

```
>>> q*q
16.0*J**2
+/-1.1313708499*J**2 (1 sigma)
```

This result is probably incorrect, since it assumes the uncertainties of the two multiplicands are uncorrelated. It would be more accurate in this case to use:

```
>>> q**2
16.0*J**2
+/-1.6*J**2 (1 sigma)
```

There is an entire subpackage dedicated to physical constants. The values of all the constants are taken from values published by the National Institute of Standards and Technology at http://physics.nist.gov/constants . Most physical constants have some form of uncertainty, which has also been published by NIST. All uncertainties are one standard deviation. There are lots of constants and quantities includes them all (with one exception: F*, the Faraday constant for conventional electrical current, which is defined in units of C_90, for which I have not found a hard reference value). Physical constants are sort of similar to compound units, for example:

```
>>> print pq.constants.proton_mass
1 m_p (proton_mass)
>>> print pq.constants.proton_mass.simplified
1.672621637e-27 kg
+/-8.3e-35 kg (1 sigma)
```

A Latex representation of the dimensionality may be obtained in the following fashion:

```
>>> g = pq.Quantity(9.80665,'m/s**2')
>>> mass = 50 * pq.kg
>>> weight = mass*g
>>> print weight.dimensionality.latex
$\mathrm{\frac{kg{\cdot}m}{s^{2}}}$
>>> weight.units = pq.N
>>> print weight.dimensionality.latex
$\mathrm{N}$
```

The Latex output is compliant with the MathText subset used by Matplotlib. To add formatted units to the axis label of a Matplotlib figure, one could use:

```
>>> ax.set_ylabel('Weight ' + weight.dimensionality.latex)
```

Greater customization is available via the markup.format_units_latex function. It allows the user to modify the font, the multiplication symbol, or to encapsulate the latex string in parentheses. Due to the complexity of CompoundUnits, the latex rendering of CompoundUnits will utilize the latex frac{num}{den} construct.

Although it is not illustrated in this guide, unicode symbols can be used to provide a more compact representation of the units. This feature is disabled by default. It can be enabled by setting the following in your ~/.pythonrc.py:

```
quantities_unicode = True
```

or you can change this setting on the fly by doing:

```
from quantities import markup
markup.config.use_unicode = True # or False
```

Even when unicode is enabled, when you pass strings to designate units, they should still conform to valid python expressions.

Attention: Quantities is not a package for describing coordinate systems that require a point of reference, like positions on a map. In particular, Quantities does not support absolute temperature scales. Instead, temperatures are assumed to be temperature *differences*. For example:

```
>>> T = 20 * pq.degC
>>> print T.rescale('K')
20.0 K
```

Proper support of coordinate systems would be a fairly large undertaking and is outside the scope of this project.

Known Issues

Quantities arrays are designed to work like normal numpy arrays. However, a few operations are not yet fully functioning.

Note: In the following code examples, it's assumed that you've initiated the following imports:

```
>>> import numpy as np
>>> import quantities as pq
```

Temperature conversion

Quantities is not designed to handle coordinate systems that require a point of reference, like positions on a map or absolute temperature scales. Proper support of coordinate systems would be a fairly large undertaking and is outside the scope of this project. Furthermore, consider the following:

```
>>> T_0 = 100 * pq.K
>>> T_1 = 200 * pq.K
>>> dT = T_1-T_0
>>> dT.units = pq.degF
```

To properly support the above example, quantities would have to distinguish absolute temperatures with temperature differences. It would have to know how to combine these two different animals, etc. The quantities project has therefore elected to limit the scope to relative quantities.

As a consequence, quantities treats temperatures as a temperature difference. This is a distinction without a difference when considering Kelvin and Rankine, or transformations between the two scales, since both scales have zero offset. Temperature scales in Celsius and Fahrenheit are different and would require a non-zero offset, which is not supported in Quantities unit transformation framework.

umath functions

Many common math functions ignore the dimensions of quantities. For example, trigonometric functions (e.g. *np.sin*) suffer this fate. For these functions, quantities arrays are treated like normal arrays and the calculations proceed as normal (except that a "not implemented" warning is raised). Note, however, this behavior is not ideal since some functions should behave differently for different units. For example, you would expect *np.sin* to give different results for an angle of 1° versus an angle of 1 radian; instead, *np.sin* extracts the magnitude of the input and assumes that it is already in radians.

To properly handle quantities, use the corresponding quantities functions whenever possible. For example, *pq.sin* will properly handle the angle inputs described above. For an exhaustive list, see the functions defined in *pq.umath*.

Functions which ignore/drop units

There are additional numpy functions not in *pq.umath* that ignore and drop units. Below is a list known functions in this category

- *vstack*
- *interp*

License

Quantities only uses BSD compatible code. See the Open Source Initiative licenses page for details on individual licenses.

License Agreement for Quantities

License Agreement for Scimath

WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Credits

Quantities was written by Darren Dale, with the hope that additional developers will contribute to — and take credit for — its development.

Special thanks to those who have made valuable contributions (roughly in order of first contribution by date)

Charles Doutriaux wrote the python wrapper for unit conversions with the UDUnits library (no longer used)

Enthought Inc. the original units registry was developed at Enthought for the Enthought Tool Suite.

John Salvatier added support for the Quantity iterator, contributed to support for simplified representations of units, comparison operators, and contributed unit tests.

Tony Yu contributed several bug fixes and contributed to the documentation.

CHAPTER 2

Developer's Guide

Documenting Quantities

Getting started

The documentation for Quantities is generated from ReStructured Text using the Sphinx documentation generation tool and the numpydoc Sphinx extension. Sphinx-0.6.3 or later is required. You can obtain Sphinx and numpydoc from the Python Package Index or by doing:

```
easy_install sphinx
```

The documentation sources are found in the `doc/` directory in the trunk. The output produced by Sphinx can be configured by editing the `conf.py` file located in the `doc/` directory. To build the users guide in html format, run from the main quantities directory:

```
python setup.py build_sphinx
```

and the html will be produced in *build/sphinx/html*. To build the pdf file:

```
python setup.py build_sphinx -b latex
cd build/sphinx/latex
make all-pdf
```

Organization of Quantities' documentation

The actual ReStructured Text files are kept in `doc`. The main entry point is `doc/index.rst`, which pulls in the `index.rst` file for the user guide and the developers guide. The documentation suite is built as a single document in order to make the most effective use of cross referencing, we want to make navigating the Quantities documentation as easy as possible.

Additional files can be added to the various guides by including their base file name (the .rst extension is not necessary) in the table of contents. It is also possible to include other documents through the use of an include statement, such as:

```
.. include:: ../TODO
```

Formatting

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx in general. Since quantities is so closely coupled with the numpy package, quantities will conform to numpy's documentation standards and use numpy's documentation tools. Please familiarize yourself with the docstring standard and the examples like these.

Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Quantities' documentation makes heavy use of cross-referencing and other semantic markup. For example, when referring to external files, use the `:file:` directive.
- Function arguments and keywords should be referred to using the *emphasis* role. This will keep Quantities' documentation consistant with Python's documentation:

  ```
  Here is a description of *argument*
  ```

  Please do not use the *default role*:

  ```
  Please do not describe `argument` like this.
  ```

  nor the `literal` role:

  ```
  Please do not describe ``argument`` like this.
  ```

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables can not be used when documenting Quantities.
- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For example:

  `:math:`\sin(x_n^2)`` yields: $\sin(x_n^2)$, and:

  ```
  .. math::

    \int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}
  ```

  yields:

  $$\int_{-\infty}^{\infty} \frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}$$

- Footnotes[1] can be added using `[#]_`, followed later by:

  ```
  .. rubric:: Footnotes

  .. [#]
  ```

- Use the *note* and *warning* directives, sparingly, to draw attention to important comments:

  ```
  .. note::
     Here is a note
  ```

[1] For example.

yields:

Note: here is a note

---

also:

Warning: here is a warning

- Use the *deprecated* directive when appropriate:

```
.. deprecated:: 0.98
   This feature is obsolete, use something else.
```

yields:

Deprecated since version 0.98: This feature is obsolete, use something else.

- Use the *versionadded* and *versionchanged* directives, which have similar syntax to the *deprecated* role:

```
.. versionadded:: 0.98
   The transforms have been completely revamped.
```

New in version 0.98: The transforms have been completely revamped.

- Use the *seealso* directive, for example:

```
.. seealso::

   Using ReST :ref:`emacs-helpers`:
      One example

   A bit about :ref:`referring-to-quantities-docs`:
      One more
```

yields:

See also:

Using ReST *Emacs helpers*: One example

A bit about *Referring to quantities documents*: One more

- The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.

Docstrings

In addition to the aforementioned formatting suggestions:

- Please limit the text width of docstrings to 70 characters.
- Keyword arguments should be described using a definition list.

Figures

Dynamically generated figures

The top level `doc` dir has a folder called `pyplots` in which you should include any pyplot plotting scripts that you want to generate figures for the documentation. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to insure that the code that is included runs and produces the advertised figure. Several figures will be saved with the same basnename as the filename when the documentation is generated (low and high res PNGs, a PDF). Quantities includes a Sphinx extension (`sphinxext/plot_directive.py`) for generating the images from the python script and including either a png copy for html or a pdf for latex:

```
.. plot:: pyplot_simple.py
   :include-source:
```

The `:scale:` directive rescales the image to some percentage of the original size, though we don't recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling. `:include-source:` will present the contents of the file, marked up as source code.

Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the `doc/pyplots/make.py` script when adding such figures, and commit the script **and** the images to svn. Please also add a line to the README in doc/pyplots for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

```
.. plot:: tex_unicode_demo.py
   :include-source
```

Referring to quantities documents

In the documentation, you may want to include to a document in the Quantities src, e.g. a license file or an example. When you include these files, include them using the `literalinclude` directive:

```
.. literalinclude:: ../examples/some_example.py
```

Internal section references

To maximize internal consistency in section labeling and references, use hypen separated, descriptive labels for section references, eg:

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:

```
See :ref:`howto-webapp`
```

Keep in mind that we may want to reorganize the contents later, so let's avoid top level names in references like `user` or `devel` or `faq` unless necesssary, because for example the FAQ "what is a backend?" could later become part of the users guide, so the label:

```
.. _what-is-a-backend
```

is better than:

```
.. _faq-backend
```

In addition, since underscores are widely used by Sphinx itself, let's prefer hyphens to separate words.

Section names, etc

For everything but top level chapters, please use `Upper lower` for section titles, eg `Possible hangups` rather than `Possible Hangups`

Emacs helpers

There is an emacs mode rst.el which automates many important ReST tasks like building and updateing table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
      (append '(("\\.txt$" . rst-mode)
                ("\\.rst$" . rst-mode)
                ("\\.rest$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

```
C-c TAB - rst-toc-insert

  Insert table of contents at point

C-c C-u - rst-toc-update

    Update the table of contents at point

C-c C-l rst-shift-region-left

    Shift region to the left

C-c C-r rst-shift-region-right

    Shift region to the right
```

Development

Quantities development uses the principles of test-driven development. New features or bug fixes need to be accompanied by unit tests based on Python's unittest package. Unit tests can be run with the following:

```
python setup.py test
```

This works with the version of unittest provided by the python-2.7 and python-3.2+ standard library.

Releases

Creating Source Releases

Quantities is distributed as a source release for Linux and OS-X. To create a source release, just do:

```
python setup.py register
python setup.py sdist --formats=zip,gztar upload --sign
```

This will create the tgz source file and upload it to the Python Package Index. Uploading to PyPi requires a .pypirc file in your home directory, something like:

```
[server-login]
username: <username>
password: <password>
```

You can create a source distribution without uploading by doing:

```
python setup.py sdist
```

This creates a source distribution in the *dist/* directory.

Creating Windows Installers

We distribute binary installers for the windows platform. In order to build the windows installer, open a DOS window, cd into the quantities source directory and run:

```
python setup.py build
python setup.py bdist_msi
```

This creates the executable windows installer in the *dist/* directory.

Building Quantities documentation

The Quantities documentation is automatically built on readthedocs.io.

Should you need to build the documentation locally, Sphinx, LaTeX (preferably TeX-Live), and dvipng are required. Once these are installed, do:

```
cd doc
make html
```

which will produce the html output and save it in build/sphinx/html. Then run:

```
make latex
cd build/latex
make all-pdf
cp Quantities.pdf ../html
```

which will generate a pdf file in the latex directory.

Python Module Index

q

Index

Q