QuantiPhy Documentation

Release 2.1.0

Ken Kundert

Jul 31, 2017

Contents

Version: 2.1.0
Released: 2017-07-30
Please post all bugs and suggestions at Github (or contact me directly at quantiphy@nurdletech.com).

CHAPTER 1

What?

QuantiPhy is a Python library that offers support for physical quantities. A quantity is the pairing of a number and a unit of measure that indicates the amount of some measurable thing. *QuantiPhy* provides quantity objects that keep the units with the number, making it easy to share them as single object. They subclass float and so can be used anywhere a number is appropriate.

CHAPTER 2

Why?

QuantiPhy naturally supports SI scale factors, which are widely used in science and engineering. SI scale factors make it possible to cleanly represent both very large and very small quantities in a form that is both easy to read and write. While generally better for humans, no general programming language provides direct support for reading or writing quantities with SI scale factors, making it difficult to write software that communicates effectively with humans. *QuantiPhy* addresses this deficiency, making it natural and simple to both input and output physical quantities.

CHAPTER 3

Features

- Flexibly reads amounts with units and SI scale factors.
- Quantities subclass the *float* class and so can be used as conventional numbers.
- Generally includes the units when printing or converting to strings and by default employs SI scale factors.
- Flexible unit conversion and scaling is supported to make it easy to convert to or from any required form.
- Provides a small but extensible collection of physical constants.

CHAPTER 4

Alternatives

There are a considerable number of Python packages dedicated to units and quantities (alternatives). However, as a rule, they focus on the units rather than the scale factors. In particular, they build a system of units that you are expected to use throughout your calculations. These packages demand a high level of commitment from their users and in turn provide unit consistency and built-in unit conversions. In contrast, *QuantiPhy* treats units basically as documentation. They are simply strings that are attached to quantities largely so they can be presented to the user when the values are printed. As such, *QuantiPhy* is a light-weight package that demands little from the user. It is used when inputting and outputting values, and then only when it provides value. As a result, it provides a simplicity in use that cannot be matched by the other packages.

CHAPTER 5

Quick Start

Install with:

```
pip3 install quantiphy
```

Requires Python3.3 or better. Python2.7 is also supported, however support for unicode is weak.

You use *Quantity* to convert numbers and units in various forms to quantities:

```
>>> from quantiphy import Quantity

>>> Tclk = Quantity(10e-9, 's')
>>> print(Tclk)
10 ns

>>> Fhy = Quantity('1420.405751786 MHz')
>>> print(Fhy)
1.4204 GHz

>>> Rsense = Quantity('1e-4Ω')
>>> print(Rsense)
100 uΩ

>>> cost = Quantity('$11_200_000')
>>> print(cost)
$11.2M

>>> Tboil = Quantity('212 °F', scale='°C')
>>> print(Tboil)
100 °C
```

Once you have a quantity, there are a variety of ways of accessing aspects of the quantity:

```
>>> Tclk.real
1e-08
```

```
>>> float(Fhy)
1420405751.786

>>> 2*cost
22400000.0

>>> Rsense.units
'Ω'

>>> str(Tboil)
'100 °C'
```

You can use the render method to flexibly convert the quantity to a string:

```
>>> Tclk.render()
'10 ns'

>>> Tclk.render(show_units=False)
'10n'

>>> Tclk.render(show_units=False, show_si=False)
'10e-9'

>>> Fhy.render(prec=8)
'1.42040575 GHz'

>>> Tboil.render(scale='°F')
'212 °F'
```

You can use the string format method or the new format strings to flexibly incorporate quantity values into strings:

```
>>> f'{Fhy}'
'1.4204 GHz'

>>> f'{Fhy:.6}'
'1.420406 GHz'

>>> f'|{Fhy:<15.6}|'
'|1.420406 GHz   |'

>>> f'|{Fhy:>15.6}|'
'|   1.420406 GHz|'

>>> f'Boiling point of water: {Tboil:s}'
'Boiling point of water: 100 °C'

>>> f'Boiling point of water: {Tboil:s°F}'
'Boiling point of water: 212 °F'
```

CHAPTER 6

Documentation

Users' Guide

Overview

QuantiPhy adds support for quantities to Python. Quantities are little more than a number combined with its units. They are used to represent physical quantities. For example, your height and weight are both quantities, having both a value and units, and both are important. For example, if I told you that Mariam's weight was 8, you might assume pounds as the unit of measure if you lived in the US and think Mariam was an infant, or you might assume stones as the units if you live in the UK and assume that she was an adult, or you might assume kilograms if you lived anywhere else and assume she was a small child. The units are very important, and in general it is always best to keep the unit of measure with the number and present the complete value when working with quantities. To do otherwise invites confusion. Just ask NASA. Readers often stumble on numbers without units as they mentally try to determine the units from context. Quantity values should be treated in a manner similar to money, which is also a quantity. Monetary amounts are almost always given with their units (a currency symbol).

Having a single object represent a quantity in a programming language is useful because it binds the units to the number making it more likely that the units will be presented with the number. In addition, quantities from *QuantiPhy* provide another important benefit. They naturally support the SI scale factors, which for those that are familiar with them are much easer to read and write than the alternatives. The most common SI scale factors are:

T (10^{12}) tera
G (10^{9}) giga
M (10^{6}) mega
k (10^{3}) kilo
m (10^{-3}) milli
μ (10^{-6}) micro
n (10^{-9}) nano
p (10^{-12}) pico
f (10^{-15}) fempto
a (10^{-18}) atto

Numbers with SI scale factors are commonly used in science and engineering to represent physical quantities because it is easy to read and write numbers both large and small. For example, the distance between the atoms in a silicon lattice is roughly 230 pm whereas the distance to the sun is about 150 Gm. Unfortunately, computers do not normally use SI scale factors. Instead, they use E-notation. The two distances would be written as 2.3e-10 m and 1.5e+11 m. Virtually all computer languages such as Python both read and write numbers in E-notation, but none naturally read or write numbers that use SI scale factors, even though SI is an international standard that has been in place for over 50 years and is widely used.

QuantiPhy is an attempt to address both of these deficiencies. It allows quantities to be represented with a single object that allows the complete quantity to be easily read or written as a single unit. It also naturally supports SI scale factors. As such, *QuantiPhy* allows computers to communicate more naturally with humans, particularly scientists and engineers.

Quantities

QuantiPhy is a library that adds support to Python for both reading and writing numbers with SI scale factors and units. The primary working construct for *QuantiPhy* is *quantiphy.Quantity*, which is a class whose objects hold the number and units that are used to represent a physical quantity. For example, to create a quantity from a string you can use:

```
>>> from quantiphy import Quantity

>>> distance_to_sun = Quantity('150 Gm')
>>> distance_to_sun.real
150000000000.0

>>> distance_to_sun.units
'm'

>>> print(distance_to_sun)
150 Gm
```

Now *distance_to_sun* contains two items, the number 150000000000.0 and the units 'm'. The 'G' was interpreted as the *giga* scale factor, which scales by 10^9.

quantiphy.Quantity is a subclass of float, and so *distance_to_sun* can be used just like any real number. For example, you can convert the distance to miles using:

```
>>> distance_in_miles = distance_to_sun / 1609.34
>>> print(distance_in_miles)
93205910.49747102
```

When printed or converted to strings quantities naturally use SI scale factors. For example, you can clean up that distance in miles using:

```
>>> distance_in_miles = Quantity(distance_to_sun / 1609.34, 'miles')
>>> print(distance_in_miles)
93.206 Mmiles
```

However, you need not explicitly do the conversion yourself. *QuantiPhy* provides many of the most common conversions for you:

```
>>> distance_in_miles = Quantity(distance_to_sun, scale='miles')
>>> print(distance_in_miles)
93.206 Mmiles
```

Specifying a Quantity Value

Normally, creating a quantity takes one or two arguments. The first is taken to be the value, and the second, if given, is taken to be the model, which is a source of default values. The value may be given as a float, as a string, or as a quantity. The string may be the name of a known constant or it may represent a number. If the string represents a number, it may be in floating point notation, in E-notation (ex: 1.2e+3), or use SI scale factors. It may also include the units. And like Python in general, the numbers may include underscores to make them easier to read (they are ignored). For example, any of the following ways can be used to specify 1ns:

```
>>> period = Quantity(1e-9, 's')
>>> print(period)
1 ns

>>> period = Quantity('0.000_000_001 s')
>>> print(period)
1 ns

>>> period = Quantity('1e-9s')
>>> print(period)
1 ns

>>> period = Quantity('1ns')
>>> print(period)
1 ns

>>> period2 = Quantity(period)
>>> print(period2)
1 ns
```

If given as a string, the value may also be the name of a known *constant*:

```
>>> k = Quantity('k')
>>> q = Quantity('q')
>>> print(k, q, sep='\n')
13.806e-24 J/K
160.22e-21 C
```

The following constants are pre-defined: h, , k, q, c, $0°C'$, ϵ_0, μ_0, and Z_0. You may add your own constants.

Currency units ($£€) are a bit different than other units, they are placed at the front of the quantity.

```
>>> print(Quantity('$11_200_000'))
$11.2M

>>> print(Quantity(11.2e6, '$'))
$11.2M
```

When using currency units, if the number has a sign, it should precede the units:

```
>>> print(Quantity('-$11_200_000'))
-$11.2M

>>> print(Quantity(-11.2e6, '$'))
-$11.2M
```

When given as a string, the number may use any of the following scale factors (though you can use the *input_sf* preference to prune this list if desired):

Y (10^{24}) yotta

Z (10^{21}) zetta
E (10^{18}) exa
P (10^{15}) peta
T (10^{12}) tera
G (10^{9}) giga
M (10^{6}) mega
k (10^{3}) kilo
_ (1)
c (10^{-2}) centi
m (10^{-3}) milli
u (10^{-6}) micro
μ (10^{-6}) micro
n (10^{-9}) nano
p (10^{-12}) pico
f (10^{-15}) fempto
a (10^{-18}) atto
z (10^{-21}) zepto
y (10^{-24}) yocto

When specifying the value as a string you may also give a name and description, and if you do they become available as the attributes *name* and *desc*. This conversion is under the control of the *assign_rec* preference. The default version of *assign_rec* accepts either '=' or ':' to separate the name from the value, and either '–', '#', or '//' to separate the value from the description if a description is given. Thus, by default *QuantiPhy* recognizes specifications of the following forms:

```
<name> = <value>
<name> = <value> -- <description>
<name> = <value> # <description>
<name> = <value> // <description>
<name>: <value>
<name>: <value> -- <description>
<name>: <value> # <description>
<name>: <value> // <description>
```

For example:

```
>>> period = Quantity('Tclk = 10ns -- clock period')
>>> print(f'{period.name} = {period}  # {period.desc}')
Tclk = 10 ns  # clock period
```

If you only specify a real number for the value, then the units, name, and description do not get values. Even if given as a string or quantity the value may not contain these extra attributes. This is where the second argument, the model, helps. It may be another quantity or it may be a string. Any attributes that are not provided by the first argument are taken from the second if available. If the second argument is a string, it is split. If it contains one value, that value is taken to be the units, if it contains two, those values are taken to be the name and units, and it it contains more than two, the remaining values are taken to be the description. If the model is a quantity, only the units are inherited. For example:

```
>>> out_period = Quantity(10*period, period)
>>> print(out_period)
100 ns

>>> freq = Quantity(100e6, 'Hz')
>>> print(freq)
```

```
100 MHz

>>> freq = Quantity(100e6, 'Fin Hz')
>>> print(f'{freq.name} = {freq}')
Fin = 100 MHz

>>> freq = Quantity(100e6, 'Fin Hz input frequency')
>>> print(f'{freq.name} = {freq} -- {freq.desc}')
Fin = 100 MHz -- input frequency
```

In addition, you can explicitly specify the units, the name, and the description using named arguments. These values override anything specified in the value or the model.

```
>>> out_period = Quantity(
...     10*period, period, name='output period',
...     desc='period at output of frequency divider'
... )
>>> print(f'{out_period.name} = {out_period} -- {out_period.desc}')
output period = 100 ns -- period at output of frequency divider
```

Finally, you can overwrite the quantity's attributes to override the units, name, or description.

```
>>> out_period = Quantity(10*period)
>>> out_period.units = 's'
>>> out_period.name = 'output period'
>>> out_period.desc = 'period at output of frequency divider'
>>> print(f'{out_period.name} = {out_period} -- {out_period.desc}')
output period = 100 ns -- period at output of frequency divider
```

Scaling When Creating a Quantity

Quantities tend to be used primarily when reading and writing numbers, and less often when processing numbers. Often data comes in an undesirable form. For example, imagine data that has been normalized to kilograms but the numbers themselves have neither units or scale factors. *QuantiPhy* allows you to scale the number and assign the units when creating the quantity:

```
>>> mass = Quantity('2.529', scale=1000, units='g')
>>> print(mass)
2.529 kg
```

In this case the value is given in kilograms, and is converted to the base units of grams by multiplying the given value by 1000. This can also be expressed as follows:

```
>>> mass = Quantity('2.529', scale=(1000, 'g'))
>>> print(mass)
2.529 kg
```

You can also specify a function to do the conversion, which is helpful when the conversion is not linear:

```
>>> def from_dB(value, units=''):
...     return 10**(value/20), units[2:]

>>> Quantity('-100 dBV', scale=from_dB)
Quantity('10 uV')
```

The conversion can also often occur if you simply state the units you wish the quantity to have:

```
>>> Tboil = Quantity('212 °F', scale='K')
>>> print(Tboil)
373.15 K
```

This assumes that the initial value is specified with units. If not, you need to provide them for this mechanism to work.

```
>>> Tboil = Quantity('212', '°F', scale='K')
>>> print(Tboil)
373.15 K
```

To do this conversion, *QuantiPhy* examines the given units (°F) and the desired units (K) and choses the appropriate converter. *QuantiPhy* provides a collection of pre-defined converters for common units:

K:	K, F, °F, R, °R
C, °C:	K, C, °C, F, °F, R, °R
m:	km, m, cm, mm, um, μm, micron, nm, Å, angstrom, mi, mile, miles
g:	oz, lb, lbs
s:	s, sec, min, hour, hr , day
b:	B

You can also create your own converters using *quantiphy.UnitConversion*:

```
>>> from quantiphy import UnitConversion

>>> UnitConversion('m', 'pc parsec', 3.0857e16)
<...>

>>> d_sol = Quantity('5 μpc', scale='m')
>>> print(d_sol)
154.28 Gm
```

This unit conversion says, when converting units of 'm' to either 'pc' or 'parsec' multiply by 3.0857e16, when going the other way, divide by 3.0857e16.

```
>>> d_sol = Quantity('154.285 Gm', scale='pc')
>>> print(d_sol)
5 upc
```

When using unit conversions it is important to only convert to units without scale factors (such as those in the first column above) when creating a quantity. For example, it is better to convert to 'm' rather than 'cm'. If the desired units used when creating a quantity includes a scale factor, then it is easy to end up with two scale factors when converting the number to a string (ex: 1 mkm or one milli-kilo-meter).

Here is an example that uses quantity rescaling. Imagine that a table is being read that gives temperature versus time, but the temperature is given in °F and the time is given in minutes, but for the purpose of later analysis it is desired that the values be converted to the more natural units of Kelvin and seconds:

```
>>> rawdata = '0 450, 10 400, 20 360'
>>> data = []
>>> for pair in rawdata.split(','):
...     time, temp = pair.split()
...     time = Quantity(time, 'min', scale='s')
...     temp = Quantity(temp, '°F', scale='K')
...     data += [(time, temp)]

>>> for time, temp in data:
...     print(f'{time:7s} {temp}')
0 s     505.37 K
```

```
600 s   477.59 K
1.2 ks  455.37 K
```

Accessing Quantity Values

There are a variety of ways of accessing the value of a quantity. If you are just interested in its numeric value, you access it with:

```
>>> h_line = Quantity('1420.405751786 MHz')

>>> h_line.real
1420405751.786

>>> float(h_line)
1420405751.786
```

Or you can use a quantity in the same way that you would use any real number, meaning that you can use it in expressions and it will evaluate to its numeric value:

```
>>> import math
>>> sagan_freq = math.pi * h_line
>>> print(sagan_freq)
4462336274.927585

>>> sagan = Quantity(sagan_freq, h_line)
>>> print(sagan)
4.4623 GHz

>>> type(h_line)
<class 'quantiphy.Quantity'>

>>> type(sagan_freq)
<class 'float'>

>>> type(sagan)
<class 'quantiphy.Quantity'>
```

Notice that when performing arithmetic operations on quantities the units are completely ignored and do not propagate in any way to the newly computed result.

If you are interested in the units of a quantity, you can use:

```
>>> h_line.units
'Hz'
```

Or you can access both the value and the units, either as a tuple or in a string:

```
>>> h_line.as_tuple()
(1420405751.786, 'Hz')

>>> str(h_line)
'1.4204 GHz'
```

SI scale factors are used by default when converting numbers to strings. The following scale factors could be used:

Y (10^{24}) yotta
Z (10^{21}) zetta

E (10^{18}) exa
P (10^{15}) peta
T (10^{12}) tera
G (10^{9}) giga
M (10^{6}) mega
k (10^{3}) kilo
m (10^{-3}) milli
u (10^{-6}) micro
n (10^{-9}) nano
p (10^{-12}) pico
f (10^{-15}) fempto
a (10^{-18}) atto
z (10^{-21}) zepto
y (10^{-24}) yocto

However, only the scale factors listed in the *output_sf* preference are actually used, and by default that is set to 'TGMkmunpfa', which avoids the more uncommon scale factors.

The render() method allows you to control the process of converting a quantity to a string. For example:

```
>>> h_line.render()
'1.4204 GHz'

>>> h_line.render(show_si=False)
'1.4204e9 Hz'

>>> h_line.render(show_units=False)
'1.4204G'

>>> h_line.render(show_si=False, show_units=False)
'1.4204e9'

>>> h_line.render(prec=6)
'1.420406 GHz'
```

show_label allows you to display the name and description of the quantity when rendering. If *show_label* is *False*, the quantity is not labeled with the name or description. Otherwise the quantity is labeled under the control of the *show_label* value and the *show_desc*, *label_fmt* and *label_fmt_full* preferences (described further in *Preferences* and *quantiphy.Quantity.set_prefs()*). If *show_label* is 'a' (for abbreviated) or if the quantity has no description, *label_fmt* is used to label the quantity with its name. If *show_label* is 'f' (for full), *label_fmt_full* is used to label the quantity with its name and description. Otherwise *label_fmt_full* is used if *show_desc* is True and *label_fmt* otherwise.

```
>>> freq.render(show_label=True)
'Fin = 100 MHz'

>>> freq.render(show_label='f')
'Fin = 100 MHz -- input frequency'

>>> Quantity.set_prefs(show_desc=True)
>>> freq.render(show_label=True)
'Fin = 100 MHz -- input frequency'

>>> freq.render(show_label='a')
'Fin = 100 MHz'
```

You can also access the full precision of the quantity:

```
>>> h_line.render(prec='full')
'1.420405751786 GHz'

>>> h_line.render(show_si=False, prec='full')
'1.420405751786e9 Hz'
```

Full precision implies whatever precision was used when specifying the quantity if it was specified as a string and if the *keep_components* preference is True. Otherwise a fixed number of digits, specified in the *full_prec* preference, is used (default=12). Generally one uses 'full' when generating output that is intended to be read by a machine.

Scaling When Rendering a Quantity

Once it comes time to output quantities from your program, you may again may be constrained in the way the numbers must be presented. *QuantiPhy* also allows you to rescale the values as you render them to strings. In this case, the value of the quantity itself remains unchanged. For example, imagine having a quantity in grams and wanting to present it in either kilograms or in pounds:

```
>>> m = Quantity('2529 g')
>>> print('mass (kg): %s' % m.render(show_units=False, scale=0.001))
mass (kg): 2.529

>>> print(m.render(scale=(0.0022046, 'lb'), show_si=False))
5.5754 lb
```

As before, functions can also be used to do the conversion. Here is an example where that comes in handy: a logarithmic conversion to dBV is performed.

```
>>> import math
>>> def to_dB(value, units):
...     return 20*math.log10(value), 'dB'+units

>>> T = Quantity('100mV')
>>> print(T.render(scale=to_dB))
-20 dBV
```

Finally, you can also use either the built-in converters or the converters you created to do the conversion simply based on the units:

```
>>> print(m.render(scale='lb'))
5.5755 lb
```

In an earlier example the units of time and temperature data were converted to normal SI units. Presumably this makes processing easier. Now, when producing the output, the units can be converted back to the original units if desired:

```
>>> for time, temp in data:
...     print('%-7s %s' % (time.render(scale='min'), temp.render(scale='°F')))
0 min   450 °F
10 min  400 °F
20 min  360 °F
```

String Formatting

Quantities can be passed into the string *format* method:

```
>>> print('{}'.format(h_line))
1.4204 GHz

>>> print('{:s}'.format(h_line))
1.4204 GHz
```

In these cases the preferences for SI scale factors, units, and precision are honored.

You can override the precision as part of the format specification

```
>>> print('{:.6}'.format(h_line))
1.420406 GHz
```

You can also specify the width and alignment.

```
>>> print('|{:15.6}|'.format(h_line))
|1.420406 GHz    |

>>> print('|{:<15.6}|'.format(h_line))
|1.420406 GHz    |

>>> print('|{:>15.6}|'.format(h_line))
|    1.420406 GHz|
```

The 'q' type specifier can be used to explicitly indicate that both the number and the units are desired and that SI scale factors should be used, regardless of the current preferences.

```
>>> print('{:.6q}'.format(h_line))
1.420406 GHz
```

Alternately, 'r' can be used to indicate just the number represented using SI scale factors is desired, and the units should not be included.

```
>>> print('{:r}'.format(h_line))
1.4204G
```

You can also use the floating point format type specifiers:

```
>>> print('{:f}'.format(h_line))
1420405751.7860

>>> print('{:e}'.format(h_line))
1.4204e+09

>>> print('{:g}'.format(h_line))
1.4204e+09
```

Use 'u' to indicate that only the units are desired:

```
>>> print('{:u}'.format(h_line))
Hz
```

Access the name or description of the quantity using 'n' and 'd'.

```
>>> print('{:n}'.format(freq))
Fin
```

```
>>> print('{:d}'.format(freq))
input frequency
```

Using the upper case versions of the format codes that print the numerical value of the quantity (SQRFEG) indicates that the quantity should be labeled with its name and perhaps its description (as if the *show_label* preference were set). They are under the control of the *show_desc*, *label_fmt* and *label_fmt_full* preferences (described further in *Preferences* and `quantiphy.Quantity.set_prefs()`).

If *show_desc* is False or the quantity does not have a description, then *label_fmtl* use used to add the labeling.

```
>>> Quantity.set_prefs(show_desc=False)
>>> trise = Quantity('10ns', name='trise')

>>> print('{:S}'.format(trise))
trise = 10 ns

>>> print('{:Q}'.format(trise))
trise = 10 ns

>>> print('{:R}'.format(trise))
trise = 10n

>>> print('{:F}'.format(trise))
trise = 0.0000

>>> print('{:E}'.format(trise))
trise = 1.0000e-08

>>> print('{:G}'.format(trise))
trise = 1e-08

>>> print('{0:n} = {0:q} ({0:d})'.format(freq))
Fin = 100 MHz (input frequency)

>>> print('{:S}'.format(freq))
Fin = 100 MHz
```

If *show_desc* is True and the quantity has a description, then *label_fmt_full* is used if the quantity has a description.

```
>>> Quantity.set_prefs(show_desc=True)

>>> print('{:S}'.format(trise))
trise = 10 ns

>>> print('{:S}'.format(freq))
Fin = 100 MHz -- input frequency
```

Finally, you can add units after the format code, which causes the number to be scaled to those units if the transformation represents a known unit conversion.

```
>>> Tboil = Quantity('Boiling point = 100 °C')
>>> print('{:S°F}'.format(Tboil))
Boiling point = 212 °F

>>> eff_channel_length = Quantity('leff = 14nm')
>>> print(f'{eff_channel_length:SÅ}')
leff = 140 Å
```

This feature can be used to simplify the conversion of the time and temperature information back into the original units:

```
>>> for time, temp in data:
...     print(f'{time:<7smin} {temp:s°F}')
0 min   450 °F
10 min  400 °F
20 min  360 °F
```

Physical Constants

QuantiPhy has several built-in constants that are available by specifying their name to the *quantiphy.Quantity* class. The following quantities are built in:

Name	MKS value	CGS value	Description
h	6.626070040e-34 J-s	6.626070040e-27 erg-s	Plank's constant
hbar,	1.054571800e-34 J-s	1.054571800e-27 erg-s	Reduced Plank's constant
k	1.38064852e-23 J/K	1.38064852e-16 erg/K	Boltzmann's constant
q	1.6021766208e-19 C	4.80320425e-10 Fr	Elementary charge
c	2.99792458e8 m/s	2.99792458e8 m/s	Speed of light
0C, 0°C	273.15 K	273.15 K	0 Celsius
eps0, ϵ_0	8.854187817e-12 F/m	—	Permittivity of free space
mu0, μ_0	4e-7π H/m	—	Permeability of free space
Z0, Z_0	376.730313461 Ohms	—	Characteristic impedance of free space

Constants are given in base units (*g*, *m*, etc.) rather than the natural units for the unit system (*kg*, *cm*, etc.). For example, when using the CGS unit system, the speed of light is given as 300Mm/s (rather than 30Gcm/s).

As shown, these constants are partitioned into two *unit systems*: *mks* and *cgs*. Only those constants that are associated with the active unit system and those that are not associated with any unit system are available when creating a new quantity. You can activate a unit system using *quantiphy.set_unit_system()*. Doing so deactivates the previous system. By default, the *mks* system is active.

You can create your own constants and unit systems using *quantiphy.add_constant()*:

```
>>> from quantiphy import Quantity, add_constant
>>> add_constant(Quantity("λh: 211.061140539mm // wavelength of hydrogen line"))

>>> hy_wavelength = Quantity('λh')
>>> print(hy_wavelength.render(show_label=True))
λh = 211.06 mm -- wavelength of hydrogen line
```

In this case is the name given in the quantity is used when creating the constant. You can also specify an alias as an argument to *add_constant*.

```
>>> add_constant(
...     Quantity("λh = 211.061140539mm # wavelength of hydrogen line"),
...     alias='lambda h'
... )

>>> hy_wavelength = Quantity('lambda h')
>>> print(hy_wavelength.render(show_label=True))
λh = 211.06 mm -- wavelength of hydrogen line
```

It is not necessary to specify both the name and the alias, one is sufficient, but the constant is accessible using either. Notice that the alias does not actually become part of the constant, it is only used for looking up the constant.

By default, user defined constants are not associated with a unit system, meaning that they are always available regardless of which unit system is being used. However, when creating a constant you can specify one or more unit systems for the constant. You need not limit yourself to the predefined *mks* and *cgs* unit systems. You can specify multiple unit systems either by specifying a list of strings for the unit systems, or by specifying one string that would contain more than one name once split.

```
>>> from quantiphy import Quantity, add_constant, set_unit_system

>>> add_constant(Quantity(4.80320427e-10, 'Fr'), 'q', 'esu gaussian')
>>> add_constant(Quantity(1.602176487e-20, 'abC'), alias='q', unit_systems='emu')
>>> q_mks = Quantity('q')
>>> set_unit_system('cgs')
>>> q_cgs = Quantity('q')
>>> set_unit_system('esu')
>>> q_esu = Quantity('q')
>>> set_unit_system('gaussian')
>>> q_gaussian = Quantity('q')
>>> set_unit_system('emu')
>>> q_emu = Quantity('q')
>>> set_unit_system('mks')
>>> print(q_mks, q_cgs, q_esu, q_gaussian, q_emu, sep='\n')
160.22e-21 C
480.32 pFr
480.32 pFr
480.32 pFr
16.022e-21 abC
```

Preferences

QuantiPhy supports a wide variety of preferences that control its behavior. For example, when rendering quantities you can control the number of digits used (*prec*), whether SI scale factors are used (*show_si*), whether the units are included (*show_units*), etc. Similar preferences also control the conversion of strings into quantities, which can help disambiguate whether a suffix represents a scale factor or a unit. The list of available preferences and their descriptions are given in the description of the *quantiphy.Quantity.set_prefs()* method.

To set a preference, use the *quantiphy.Quantity.set_prefs()* class method. You can set more than one preference at once:

```
>>> Quantity.set_prefs(prec=6, map_sf={'u': 'μ'})
```

This statements tells *QuantiPhy* to use 7 digits (the *prec* plus 1) and to output μ rather u for the 10^{-6} scale factor.

Setting preferences to *None* returns them to their default values:

```
>>> Quantity.set_prefs(prec=None, map_sf=None)
```

The preferences are changed on the class itself, meaning that they affect any instance of that class regardless of whether they were instantiated before or after the preferences were set. If you would like to have more than one set of preferences, then you should subclass *quantiphy.Quantity*. For example, imagine a situation where you have different types of quantities that would naturally want different precisions:

```
>>> class Temperature(Quantity):
...     pass
>>> Temperature.set_prefs(prec=1, known_units='K', spacer='')

>>> class Frequency(Quantity):
```

```
...     pass
>>> Frequency.set_prefs(prec=5, spacer='')

>>> frequencies = []
>>> for each in '-25.3 999987.7, 25.1  1000207.1, 74.9  1001782.3'.split(','):
...     temp, freq = each.split()
...     frequencies.append((Temperature(temp, 'C'),  Frequency(freq, 'Hz')))

>>> for temp, freq in frequencies:
...     print(f'{temp:>4s}  {freq}')
-25C  999.988kHz
 25C  1.00021MHz
 75C  1.00178MHz
```

In this example, a subclass is created that is intended to report in concentrations.

```
>>> class Concentration(Quantity):
...     pass
>>> Concentration.set_prefs(
...     map_sf = dict(u=' PPM', n= ' PPB', p=' PPT'),
...     show_label = True,
... )

>>> pollutants = dict(CO=5, SO2=20, NO2=0.10)
>>> concentrations = [Concentration(v, scale=1e-6, name=k) for k, v in pollutants.
↪items()]
>>> for each in concentrations:
...     print(each)
CO = 5 PPM
SO2 = 20 PPM
NO2 = 100 PPB
```

When a subclass is created, the preferences active in the main class are copied into the subclass. Subsequent changes to the preferences in the main class do not affect the subclass.

You can also go the other way and override the preferences on a specific quantity.

```
>>> print(hy_wavelength)
211.06 mm

>>> hy_wavelength.show_label = True
>>> print(hy_wavelength)
λh = 211.06 mm -- wavelength of hydrogen line
```

This is often the way to go with quantities that have logarithmic units such as decibels (dB) or shannons (Sh) (or the related bit, digits, nats, hartleys, etc.). In these cases use of SI scale factors is often undesired.

```
>>> gain = Quantity(0.25, 'dB')
>>> print(gain)
250 mdB

>>> gain.show_si = False
>>> print(gain)
250e-3 dB
```

To retrieve a preference, use the *quantiphy.Quantity.get_pref()* class method. This is useful with *known_units*. Normally setting *known_units* overrides the existing units. You can simply add more with:

```
>>> Quantity.set_prefs(known_units=Quantity.get_pref('known_units') + ['K'])
```

A variation on *quantiphy.Quantity.set_prefs()* is *quantiphy.Quantity.prefs()*. It is basically the same, except that it is meant to work with Python's *with* statement to temporarily override preferences:

```
>>> with Quantity.prefs(show_si=False, show_units=False):
...     for time, temp in data:
...         print('%-7s %s' % (time, temp))
0       505.37
600     477.59
1.2e3   455.37

>>> print('Final temperature = %s @ %s.' % data[-1][::-1])
Final temperature = 455.37 K @ 1.2 ks.
```

Notice that the specified preferences only affected the table, not the final printed values, which were rendered outside the *with* statement.

Ambiguity of Scale Factors and Units

By default, *QuantiPhy* treats both the scale factor and the units as being optional. With the scale factor being optional, the meaning of some specifications can be ambiguous. For example, '1m' may represent 1 milli or it may represent 1 meter. Similarly, '1meter' my represent 1 meter or 1 milli-eter. In this case *QuantiPhy* gives preference to the scale factor, so '1m' normally converts to 1e-3. To allow you to avoid this ambiguity, *QuantiPhy* accepts '_' as the unity scale factor. In this way '1_m' is unambiguously 1 meter. You can instruct *QuantiPhy* to output '_' as the unity scale factor by specifying the *unity_sf* argument to *quantiphy.Quantity.set_prefs()*:

```
>>> Quantity.set_prefs(unity_sf='_', spacer='')
>>> l = Quantity(1, 'm')
>>> print(l)
1_m
```

This is often a good way to go if you are outputting numbers intended to be read by people and machines.

If you need to interpret numbers that have units and are known not to have scale factors, you can specify the *ignore_sf* preference:

```
>>> Quantity.set_prefs(ignore_sf=True, unity_sf='', spacer=' ')
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1000.0, 'm')

>>> print(l)
1 km

>>> Quantity.set_prefs(ignore_sf=False)
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1.0, '')
```

If there are scale factors that you know you will never use, you can instruct *QuantiPhy* to interpret a specific set and ignore the rest using the *input_sf* preference.

```
>>> Quantity.set_prefs(input_sf='GMk')
>>> l = Quantity('1000m')
>>> l.as_tuple()
```

```
(1000.0, 'm')

>>> print(l)
1 km
```

Specifying *input_sf=None* causes *QuantiPhy* to again accept all known scale factors again.

```
>>> Quantity.set_prefs(input_sf=None)
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1.0, '')
```

Alternatively, you can specify the units you wish to use whose leading character is a scale factor. Once known, these units no longer confuse *QuantiPhy*. These units can be specified as a list or as a string. If specified as a string the string is split to form the list. Specifying the known units replaces any existing known units.

```
>>> d1 = Quantity('1 au')
>>> d2 = Quantity('1000 pc')
>>> print(d1.render(show_si=False), d2, sep='\n')
1e-18 u
1 nc

>>> Quantity.set_prefs(known_units='au pc')
>>> d1 = Quantity('1 au')
>>> d2 = Quantity('1000 pc')
>>> print(d1.render(show_si=False), d2, sep='\n')
1 au
1 kpc
```

This same issue comes up for temperature quantities when given in Kelvin. There are again several ways to handle this. First you can specify the acceptable input scale factors leaving out 'K', ex. *input_sf* = 'TGMkmunpfa'. Alternatively, you can specify 'K' as one of the known units. Finally, if you know exactly when you will be converting a temperature to a quantity, you can specify *ignore_sf* for that specific conversion. The effect is the same either way, 'K' is interpreted as a unit rather than a scale factor.

Formatting Tabular Data

When creating tables it is often desirable to align the decimal points of the numbers, and perhaps align the units. You can use the *number_fmt* to arrange this. *number_fmt* is a format string that if specified is used to convert the components of a number into the final number. You can control the widths and alignments of the components to implement specific arrangements. *number_fmt* is passed to the string *format* function with named arguments: *whole*, *frac* and *units*, which contains the integer part of the number, the fractional part including the decimal point, and the units including the scale factor. More information about the content of the components can be found in *quantiphy.Quantity.set_prefs()*.

For example, you can align the decimal point and units of a column of numbers like this:

```
>>> lengths = [
...     Quantity(l)
...     for l in '1mm, 10mm, 100mm, 1.234mm, 12.34mm, 123.4mm'.split(',')
... ]

>>> with Quantity.prefs(number_fmt='{whole:>3s}{frac:<4s} {units}'):
...     for l in lengths:
...         print(l)
  1     mm
```

```
 10      mm
100      mm
  1.234 mm
 12.34   mm
123.4    mm
```

You can also give a function as the value for *number_fmt* rather than a string. It would be called with *whole*, *frac* and *units* as arguments given in that order. The function is expected to return the assembled number as a string. For example:

```
>>> def fmt_num(whole, frac, units):
...     return '{mantissa:<5s} {units}'.format(mantissa=whole+frac, units=units)

>>> with Quantity.prefs(number_fmt=fmt_num):
...     for l in lengths:
...         print(l)
1     mm
10    mm
100   mm
1.234 mm
12.34 mm
123.4 mm
```

If there are multiple columns it might be necessary to apply a different format to each column. In this case, it often makes sense to create a subclass of Quantity for each column that requires distinct formatting:

```
>>> def format_temperature(whole, frac, units):
...     return '{:>5s} {:<5s}'.format(whole+frac, units)

>>> class Temperature(Quantity):
...     pass
>>> Temperature.set_prefs(
...     prec = 1, known_units = 'K', number_fmt = format_temperature
... )

>>> class Frequency(Quantity):
...     pass
>>> Frequency.set_prefs(prec=5, number_fmt = '{whole:>3s}{frac:<6s} {units}')

>>> frequencies = []
>>> for each in '-25.3 999987.7, 25.1 1000207.1, 74.9 1001782.3'.split(','):
...     temp, freq = each.split()
...     frequencies.append((Temperature(temp, 'C'),  Frequency(freq, 'Hz')))

>>> for temp, freq in frequencies:
...     print(temp, freq)
  -25 C     999.988   kHz
   25 C       1.00021 MHz
   75 C       1.00178 MHz
```

Extract Quantities

It is possible to put a collection of quantities in a text string and then use the *quantiphy.Quantity.extract()* method to parse the quantities and return them in a dictionary. For example:

```
>>> design_parameters = '''
...     Fref = 156 MHz      -- Reference frequency
...     Kdet = 88.3 uA      -- Gain of phase detector
...     Kvco = 9.07 GHz/V   -- Gain of VCO
... '''
>>> quantities = Quantity.extract(design_parameters)

>>> Quantity.set_prefs(
...     label_fmt='{n} = {v}',
...     label_fmt_full='{V:<18}  # {d}',
...     show_label='f',
... )
>>> for q in quantities.values():
...     print(q)
Fref = 156 MHz      # Reference frequency
Kdet = 88.3 uA      # Gain of phase detector
Kvco = 9.07 GHz/V   # Gain of VCO
```

quantiphy.Quantity.extract() ignores blank lines and any line that does not have a value, so you can insert comments into the string by giving a description without a name or value:

```
>>> design_parameters = '''
...     -- PLL Design Parameters
...
...     Fref = 156 MHz  -- Reference frequency
...     Kdet = 88.3 uA  -- Gain of phase detector
...     Kvco = 9.07 GHz/V  -- Gain of VCO
... '''
>>> globals().update(Quantity.extract(design_parameters))

>>> print(f'{Fref:S}\n{Kdet:S}\n{Kvco:S}', sep='\n')
Fref = 156 MHz      # Reference frequency
Kdet = 88.3 uA      # Gain of phase detector
Kvco = 9.07 GHz/V   # Gain of VCO
```

In this case the output of the *quantiphy.Quantity.extract()* call is fed into globals().update() so as to add the quantities into the module namespace, making the quantities accessible as local variables.

Any number of quantities may be given, with each quantity given on its own line. The identifier given to the left '=' is the name of the variable in the local namespace that is used to hold the quantity. The text after the '–' is used as a description of the quantity.

In this example the output of *quantiphy.Quantity.extract()* is added into the local names space. This is an example of how simulation scripts could be written. The system and simulation parameters would be gathered together at the top into a multiline string, which would then be read and loaded into the local name space. It allows you to quickly give a complete description of a collection of parameters when the goal is to put something together quickly in an expressive manner.

Here is an example that uses this feature to read parameters from a file. This is basically the same idea as above, except the design parameters are kept in a separate file. It also subclasses *quantiphy.Quantity* to create a version that displays the name and description by default.

```
>>> from quantiphy import Quantity
>>> from inform import os_error, fatal, display

>>> class VerboseQuantity(Quantity):
...     show_label = 'f'
...     label_fmt = '{n} = {v}'
```

```
...     label_fmt_full = '{V:<18} -- {d}'

>>> filename = 'parameters'
>>> try:
...     with open(filename) as f:
...         globals().update(VerboseQuantity.extract(f.read()))
... except OSError as err:
...     fatal(os_error(err))
... except ValueError as err:
...     fatal(err, culprit=filename)

>>> display(Fref, Kdet, Kvco, sep='\n')
Fref = 156 MHz      -- Reference frequency
Kdet = 88.3 uA      -- Gain of phase detector (Imax)
Kvco = 9.07 GHz/V   -- Gain of VCO
```

Translating Quantities

quantiphy.Quantity.all_from_conv_fmt() recognizes conventionally formatted numbers and quantities embedded in text and reformats them using *quantiphy.Quantity.render()*. This is an difficult task in general, and so some constraints are placed on the values to make them easier to distinguish. Specifically, the units, if given, must be simple and immediately adjacent to the number. Units are simple if they only consist of letters and underscores. The characters °, Å, Ω and ℧ are also allowed. So '47e3Ohms', '50_Ohms' and '1.0e+12Ω' are recognized as quantities, but '50 Ohms' and '12m/s' are not.

Besides the text to be translated, `all_from_conv_fmt()` takes the same arguments as `render()`, though they must be given as named arguments.

```
>>> test_results = '''
... Applying stimulus @ 2.00500000e-04s: V(in) = 5.000000e-01V.
... Pass @ 3.00500000e-04s: V(out): expected=2.00000000e+00V, measured=1.
↪99999965e+00V, diff=3.46117130e-07V.
... '''.strip()

>>> Quantity.set_prefs(spacer='')
>>> translated = Quantity.all_from_conv_fmt(test_results)
>>> print(translated)
Applying stimulus @ 200.5us: V(in) = 500mV.
Pass @ 300.5us: V(out): expected=2V, measured=2V, diff=346.12nV.
```

quantiphy.Quantity.all_from_si_fmt() is similar, except that it recognizes quantities formatted with either a scale factor or units and ignores plain numbers. Again, units are expected to be simple and adjacent to their number.

```
>>> Quantity.set_prefs(spacer='')
>>> translated_back = Quantity.all_from_si_fmt(translated, show_si=False)
>>> print(translated_back)
Applying stimulus @ 200.5e-6s: V(in) = 500e-3V.
Pass @ 300.5e-6s: V(out): expected=2V, measured=2V, diff=346.12e-9V.
```

Notice in the translations the quantities lost resolution. This is avoided if you use 'full' precision:

```
>>> translated = Quantity.all_from_conv_fmt(test_results, prec='full')
>>> print(translated)
Applying stimulus @ 200.5us: V(in) = 500mV.
Pass @ 300.5us: V(out): expected=2V, measured=1.99999965V, diff=346.11713nV.
```

Equivalence

You can determine whether the value of a quantity or real number is equivalent to that of a quantity using *quantiphy.Quantity.is_close()*. The two values need not be identical, they just need to be close to be deemed equivalent. The *reltol* and *abstol* preferences are used to determine if they are close.

```
>>> h_line.is_close(h_line)
True

>>> h_line.is_close(h_line + 1)
True

>>> h_line.is_close(h_line + 1e4)
False
```

quantiphy.Quantity.is_close() returns true if the units match and if:

abs(*a* - *b*) <= max(reltol * max(abs(*a*), abs(*b*)), abstol)

where *a* and *b* represent *other* and the numeric value of the underlying quantity.

By default, *is_close()* looks at the both the value and the units if the argument has units. In this way if you compare two quantities with different units, the *is_close()* test will always fail if their units differ. This behavior can be overridden by specifying *check_units*.

```
>>> Quantity('10ns').is_close(Quantity('10nm'))
False

>>> Quantity('10ns').is_close(Quantity('10nm'), check_units=False)
True
```

Exceptional Values

QuantiPhy supports NaN (not a number) and infinite values:

```
>>> inf = Quantity('inf Hz')
>>> print(inf)
inf Hz
```

```
>>> nan = Quantity('NaN Hz')
>>> print(nan)
nan Hz
```

You can test whether the value of the quantity is infinite or is not-a-number using *quantiphy.Quantity.is_infinite()* or *quantiphy.Quantity.is_nan()*:

```
>>> h_line.is_infinite()
False

>>> inf.is_infinite()
True

>>> h_line.is_nan()
False

>>> nan.is_nan()
True
```

Exceptions

A *ValueError* is raised if *quantiphy.Quantity* is passed a string it cannot convert into a number:

```
>>> try:
...     q = Quantity('g')
... except ValueError as err:
...     print(err)
g: not a valid number.
```

A *KeyError* is raised if a unit conversion is requested but no suitable unit converter is available.

```
>>> q = Quantity('93 Mmi', scale='pc')
Traceback (most recent call last):
...
KeyError: "Unable to convert between 'pc' and 'mi'."
```

A *KeyError* is also raised if you specify an unknown preference.

```
>>> Quantity.set_prefs(precision=6)
Traceback (most recent call last):
...
KeyError: 'precision'
```

A *NameError* is raised if a constant is created without a name or if you try to set or get a preference that is not supported.

```
>>> q = add_constant(Quantity('1ns'))
Traceback (most recent call last):
...
NameError: No name specified.
```

Classes and Functions

Quantities

class quantiphy.**Quantity**
Create a Physical Quantity

A quantity is a number paired with a unit of measure.

Parameters

- **value** (*real, string or quantity*) – The value of the quantity. If a string, it may be the name of a pre-defined constant or it may be a number that may be specified with SI scale factors and/or units. For example, the following are all valid: '2.5ns', '1.7 MHz', '1e6Ω', '2.8_V', '1e4 m/s', '$10_000', '42', '', etc. The string may also have name and description if they are provided in a way recognizable by *assign_rec*. For example, 'trise: 10ns – rise time' or 'trise = 10ns # rise time' would work with the default recognizer.
- **model** (*quantity or string*) – Used to pick up any missing attibutes (*units*, *name*, *desc*). May be a quantity or a string. If model is a quantity, only its units would be taken. If model is a string, it is split. Then, if there is one item, it is taken to be *units*. If there are two, they are taken to be *name* and *units*. And if there are three or more, the first two are taken to the be *name* and *units*, and the remainder is taken to be *description*.

- **units** (*str*) – Overrides the units taken from *value* or *model*.
- **scale** (*float, tuple, func, or string):*) –
  - If a float, it multiplies by the given value to compute the value of the quantity.
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the quantity.
  - If a function, it takes two arguments, the given value and the units and it returns two values, the value and units of the quantity.
  - If a string, it is taken to the be desired units. This value along with the units of the given value are used to select a known unit conversion, which is applied to create the quantity.
- **name** (*str*) – Overrides the name taken from *value* or *model*.
- **desc** (*str*) – Overrides the desc taken from *value* or *model*.
- **ignore_sf** (*bool*) – Assume the value given within a string does not employ a scale factors. In this way, '1m' is interpreted as 1 meter rather than 1 milli.

Raises

- **KeyError** – A unit conversion was requested and there is no corresponding unit converter or assignment recognizer (*assign_rec*) does not match at least the value (*val*).
- **ValueError** – A string was passed that cannot be converted to a quantity.

You can use *Quantity* to create quantities from floats, strings, or other quantities. If a float is given, *model* or *units* would be used to specify the units.

Examples:

```
>>> from quantiphy import Quantity
>>> from math import pi, tau
>>> newline = '''
... '''

>>> fhy = Quantity('1420.405751786 MHz')
>>> sagan = Quantity(pi*fhy, 'Hz')
>>> sagan2 = Quantity(tau*fhy, fhy)
>>> print(fhy, sagan, sagan2, sep=newline)
1.4204 GHz
4.4623 GHz
8.9247 GHz
```

You can use *scale* to scale the number or convert to different units when creating the quantity.

Examples:

```
>>> Tfreeze = Quantity('273.15 K', ignore_sf=True, scale='°C')
>>> print(Tfreeze)
0 °C

>>> Tboil = Quantity('212 °F', scale='°C')
>>> print(Tboil)
100 °C
```

classmethod all_from_conv_fmt (*text*, ***kwargs*)

Convert all numbers and quantities from conventional notation.

Parameters

- **text** (*str*) – A search and replace is performed on this text. The search looks for numbers and quantities in floating point or e-notation. They are replaced with the same number rendered as a quantity. To be recognized any units must be simple (only letters or underscores, no digits or symbols) and the units must be immediately adjacent to the number.
- ****kwargs** – By default the numbers are rendered using the currently active preferences, but any valid argument to `Quantity.render()` can be passed in to control the rendering.

Returns A copy of *text* where all numbers that were formatted conventionally have been reformatted.

Return type str

Example:

```
>>> text = 'Applying stimulus @ 2.05000e-05s: V(in) = 5.00000e-01V.'
>>> with Quantity.prefs(spacer=''):
...     xlated = Quantity.all_from_conv_fmt(text)
...     print(xlated)
Applying stimulus @ 20.5us: V(in) = 500mV.
```

classmethod **all_from_si_fmt**(*text*, ***kwargs*)

Convert all numbers and quantities from SI notation.

Parameters

- **text** (*str*) – A search and replace is performed on this text. The search looks for numbers and quantities formatted in SI notation (must have either a scale factor or units or both). They are replaced with the same number rendered as a quantity. To be recognized any units must be simple (only letters or underscores, no digits or symbols) and the units must be immediately adjacent to the number.
- ****kwargs** – By default the numbers are rendered using the currently active preferences, but any valid argument to `Quantity.render()` can be passed in to control the rendering.

Returns A copy of *text* where all numbers that were formatted with SI scale factors have been reformatted.

Return type str

Example:

```
>>> print(Quantity.all_from_si_fmt(xlated))
Applying stimulus @ 20.5 us: V(in) = 500 mV.

>>> print(Quantity.all_from_si_fmt(xlated, show_si=False))
Applying stimulus @ 20.5e-6 s: V(in) = 500e-3 V.
```

as_tuple()

Returns a tuple that contains the value as a float along with its units.

Example:

```
>>> period = Quantity('10ns')
>>> period.as_tuple()
(1e-08, 's')
```

classmethod **extract**(*text*)

Extract quantities

Takes a string that contains quantity definitions, one per line, and returns those quantities in a dictionary.

Parameters **quantities** (*str*) – The string that contains the quantities, one definition per line. Each is parsed by *assign_rec*. By default, the lines are assumed to be of the form:

```
[<name> = <value>]
[<name> = <value>] [-- <description>]
[<name> = <value>] [# <description>]
[<name> = <value>] [// <description>]
[<name>: <value>]
[<name>: <value>] [-- <description>]
[<name>: <value>] [# <description>]
[<name>: <value>] [// <description>]
```

The brackets indicate that the name/value pair and the description is optional. However, <name> must be given if <value> is given.

<name>: the name is used as a key for the value.

<value>: A number with optional units (ex: 3 or 1pF or 1 kOhm), the units need not be a simple identifier (ex: 9.07 GHz/V).

<description>: Optional textual description (ex: Gain of PD (Imax)).

Blank lines and any line that does not contain a value are ignored. So with the default *assign_rec*, lines with the following form are ignored:

```
-- comment

# comment
// comment
```

Returns a dictionary of quantities for the values specified in the argument.

Return type dict

Raises **ValueError** – Value is not a valid number or was not given a name.

Example:

```
>>> sagan_frequencies = r'''
...     -- Carl Sagan's SETI frequencies of high interest
...
...     f_hy = 1420.405751786 MHz -- Hydrogen line frequency
...     f_sagan1 = 4462.336274928 MHz -- Sagan's first frequency: pi*f_hy
...     f_sagan2 = 8924.672549855 MHz -- Sagan's second frequency: 2*pi*f_hy
... '''
>>> freqs = Quantity.extract(sagan_frequencies)
>>> for f in freqs.values():
...     print(f.render(show_label='f'))
f_hy = 1.4204 GHz -- Hydrogen line frequency
f_sagan1 = 4.4623 GHz -- Sagan's first frequency: pi*f_hy
f_sagan2 = 8.9247 GHz -- Sagan's second frequency: 2*pi*f_hy

>>> globals().update(freqs)
>>> print(f_hy, f_sagan1, f_sagan2, sep=newline)
1.4204 GHz
4.4623 GHz
8.9247 GHz
```

classmethod get_pref(*name*)
Get class preference

Returns the value of given preference.

Parameters **name** (*str*) – Name of the desired preference. See `Quantity.set_prefs()` for list of preferences.

Raises **KeyError** – unknown preference.

Example:

```
>>> Quantity.set_prefs(known_units='au')
>>> known_units = Quantity.get_pref('known_units')
>>> known_units.append('pc')
>>> Quantity.set_prefs(known_units=known_units)
>>> print(Quantity.get_pref('known_units'))
['au', 'pc']
```

is_close(*other*, *reltol=None*, *abstol=None*, *check_units=True*)
Are values equivalent?

Indicates whether the value of a quantity or real number is equivalent to that of a quantity. The two values need not be identical, they just need to be close to be deemed equivalent.

Parameters

- **other** (*quantity or real*) – The value to compare against.
- **reltol** (*float*) – The relative tolerance. If not specified. the *reltol* preference is used, which defaults to 1u.
- **abstol** (*float*) – The absolute tolerance. If not specified. the *abstol* preference is used, which defaults to 1p.
- **check_units** (*bool*) – If True (the default), and if *other* is a quantity, compare the units of the two values, if they differ return False. Otherwise only compare the numeric values, ignoring the units.

Returns Returns true if `abs(a - b) <= max(reltol * max(abs(a), abs(b)), abstol)` where `a` and `b` represent *other* and the numeric value of the underlying quantity.

Return type bool

Example:

```
>>> print(
...     c.is_close(c),                    # should pass, is identical
...     c.is_close(c+1),                  # should pass, is close
...     c.is_close(c+1e4),                # should fail, not close
...     c.is_close(Quantity(c+1, 'm/s')), # should pass, is close
...     c.is_close(Quantity(c+1, 'Hz')),  # should fail, wrong units
... )
True True False True False
```

is_infinite()
Test value to determine if quantity is infinite.

Example:

```
>>> inf = Quantity('inf Hz')
>>> inf.is_infinite()
True
```

is_nan()
Test value to determine if quantity is not a number.

Example:

```
>>> nan = Quantity('NaN Hz')
>>> nan.is_nan()
True
```

static map_sf_to_greek(*sf*)
Render scale factors in Greek alphabet if appropriate.

Pass this dictionary to *map_sf* preference if you prefer μ rather than u.

Example:

```
>>> with Quantity.prefs(map_sf=Quantity.map_sf_to_greek):
...     print(Quantity('mu0').render(show_label='f'))
μ0 = 1.2566 μH/m -- permeability of free space
```

static map_sf_to_sci_notation(*sf*)
Render scale factors in scientific notation

Pass this function to *map_sf* preference if you prefer your large and small numbers in classic scientific notation. It also causes 'u' to be converted to 'μ'. Set *show_si* False to format all numbers in scientific notation.

Example:

```
>>> with Quantity.prefs(map_sf=Quantity.map_sf_to_sci_notation):
...     print(
...         Quantity('k').render(show_label='f'),
...         Quantity('mu0').render(show_label='f'),
...         sep=newline,
...     )
k = 13.806×10^24 J/K -- Boltzmann's constant
μ0 = 1.2566 μH/m -- permeability of free space
```

classmethod prefs(***kwargs*)
Set class preferences.

This is just like `Quantity.set_prefs()`, except it is designed to work as a context manager, meaning that it is meant to be used with Python's *with* statement. It allows preferences to be set to new values temporarily. They are reset upon exiting the *with* statement. For example:

```
>>> with Quantity.prefs(ignore_sf=True):
...     t = Quantity('600_000 K')
>>> t_bad = Quantity('600_000 K')
>>> print(t, t_bad, sep=newline)
600 kK
600M
```

See `Quantity.set_prefs()` for list of available arguments.

Raises **KeyError** – unknown preference.

render(*show_units=None*, *show_si=None*, *prec=None*, *show_label=None*, *scale=None*)
Convert quantity to a string.

Parameters

- **show_units** (*bool*) – Whether the units should be included in the string.
- **show_si** (*bool*) – Whether SI scale factors should be used. If true, SI scale factors are used if the appropriate scale factor is available, otherwise engineering format is used. If false, engineering format is used for all numbers. Engineering format is normal E-notation except that the exponents are constrained to be a multiple of 3.
- **prec** (*integer or 'full'*) – The desired precision (one plus this value is the desired number of digits). If specified as 'full', the full original precision is used.
- **show_label** (*'f', 'a', or boolean*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.
  - neither is used if *show_label* is False,
  - otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),
  - otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).
- **scale** (*real, pair, function, or string:*) –
  - If a float, it scales the displayed value (the quantity is multiplied by scale before being converted to the string).
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the displayed value.
  - If a function, it takes two arguments, the value and the units of the quantity and it returns two values, the value and units of the displayed value.
  - If a string, it is taken to the be desired units. This value along with the units of the quantity are used to select a known unit conversion, which is applied to create the displayed value.

Raises **KeyError** – A unit conversion was requested and there is no corresponding unit converter.

Example:

```
>>> c = Quantity('c')
>>> print(
...     c.render(),
...     c.render(show_units=False),
...     c.render(show_si=False),
...     c.render(prec=6),
...     c.render(prec='full'),
...     c.render(show_label=True),
...     c.render(show_label='f'),
...     sep=newline
... )
299.79 Mm/s
299.79M
299.79e6 m/s
299.7925 Mm/s
299.792458 Mm/s
c = 299.79 Mm/s
c = 299.79 Mm/s -- speed of light

>>> print(
```

```
...     Tfreeze.render(scale='°F'),
...     Tboil.render(scale='°F'),
...     sep=newline
... )
32 °F
212 °F
```

classmethod set_prefs (***kwargs*)

Set class preferences.

Any values not passed in are left alone. Pass in *None* to reset a preference to its default value.

Parameters

- **abstol** (*float*) – Absolute tolerance, used by `Quantity.is_close()` when determining equivalence. Default is 10^{12}.
- **assign_rec** (*str*) – Regular expression used to recognize an assignment. Used in constructor and extract(). By default an '=' or ':' separates the name from the value and a '–', '#' or '//' separates the value from the description, if a description is given. So the default recognizes the following forms:

```
'vel = 60 m/s'
'vel = 60 m/s -- velocity'
'vel = 60 m/s # velocity'
'vel = 60 m/s // velocity'
'vel: 60 m/s'
'vel: 60 m/s -- velocity'
'vel: 60 m/s # velocity'
'vel: 60 m/s // velocity'
```

  The name, value, and description are identified in the regular expression using named groups the names *name*, *val* and *desc*. For example:

```
assign_req = r'(?P<name>.*+) = (?P<val>.*?) -- (?P<desc>.*?)',
```

- **full_prec** (*int*) – Default full precision in digits where 0 corresponds to 1 digit. Must be nonnegative. This precision is used when the full precision is requested and the precision is not otherwise known. Default is 12.
- **ignore_sf** (*bool*) – Whether all scale factors should be ignored by default when recognizing numbers.
- **input_sf** (*str*) – Which scale factors to recognize when reading numbers. The default is 'YZEPTGMKk_cmuμnpfazy'. You can use this to ignore the scale factors you never expect to reduce the chance of a scale factor/unit ambiguity. For example, if you expect to encounter temperatures in Kelvin and can do without 'K' as a scale factor, you might use 'TGMK_munpfa'. This also gets rid of the unusual scale factors.
- **keep_components** (*bool*) – Indicate whether components should be kept if quantity value was given as string. Doing so takes a bit of space, but allows the original precision of the number to be recreated when full precision is requested.
- **known_units** (*list or string*) – List of units that are expected to be used in preference to a scale factor when the leading character could be mistaken as a scale factor. If a string is given, it is split at white space to form the list. When set, any previous known units are overridden.
- **label_fmt** (*str*) – Format string used when label is requested if the quantity does not have a description or if the description was not requested (if *show_desc* is False). Is passed

through string .format() method. Format string takes two possible arguments named *n* and *v* for the name and value. A typical values include:

```
'{n} = {v}'     (default)
'{n}: {v}'
```

- **label_fmt_full** (*str*) – Format string used when label is requested if the quantity has a description and the description was requested (if *show_desc* is True). Is passed through string .format() method. Format string takes four possible arguments named *n*, *v*, *d* and *V* for the name, value, description, and value as formatted by *label_fmt*. Typical value include:

```
'{n} = {v} --  {d}'     (default)
'{n} = {v} # {d}'
'{n} = {v} // {d}'
'{n}: {v} -- {d}'
'{V} -- {d}'
'{V:<20}  # {d}'
```

  The last example shows the *V* argument with alignment and width modifiers. In this case the modifiers apply to the name and value after being they are combined with the *label_fmt*. This is typically done when printing several quantities, one per line, because it allows you to line up the descriptions.

- **map_sf** (*dictionary or function*) – Use this to change the way individual scale factors are rendered, ex: map_sf={'u': 'μ'} to render micro using mu. If a function is given, it takes a single string argument, the nominal scale factor, and returns a string, the desired scale factor. *QuantiPhy* provides two predefined functions intended for use with *maps_sf*: `Quantity.map_sf_to_greek()` and `Quantity.map_sf_to_sci_notation()`.

- **number_fmt** (*dictionary or function*) – Format string used to convert the components of the number into the number itself. Normally this is not necessary. However, it can be used to perform special formatting that is helpful when aligning numbers in tables. It allows you to specify the widths and alignments of the individual components. There are three named components: *whole*, *frac*, and *units*. *whole* contains the portion of the mantissa to the left of the radix (decimal point). It is the whole mantissa if there is no radix. It also includes the sign and the leading units (currency symbols), if any. *frac* contains the radix and the fractional part. It also contains the exponent if the number has one. *units* contains the scale factor and units. The following value can be used to align both the radix and the units, and give the number a fixed width:

```
number_fmt = '{whole:>3s}{frac:<4s} {units:<3s}'
```

  The various widths and alignments could be adjusted to fit a variety of needs.

  It is also possible to specify a function as *number_fmt*, in which case it is passed the three values in order (*whole*, *frac* and *units*) and it expected to return the number as a string.

- **output_sf** (*str*) – Which scale factors to output, generally one would only use familiar scale factors. The default is 'TGMkmunpfa', which gets rid or the very large ('YZEP') and very small ('zy') scale factors that many people do not recognize.

- **prec** (*int*) – Default precision in digits where 0 corresponds to 1 digit. Must be non-negative. This precision is used when the full precision is not required. Default is 4.

- **reltol** (*float*) – Relative tolerance, used by `Quantity.is_close()` when determining equivalence. Default is 10^6.

- **show_desc** (*bool*) – Whether the description should be shown if it is available when showing the label. By default *show_desc* is False.

  Deprecated since version 2.1: Use `show_label='f'` instead.

- **show_label** (*'f', 'a', or bool*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.

  – Neither is used if *show_label* is False,

  – otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),

  – otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).

- **show_si** (*bool*) – Use SI scale factors by default. If this is not set, engineering format is used. Engineering format is normal E-notation except that the exponents are constrained to be a multiple of 3.

- **spacer** (*str*) – The spacer text to be inserted in a string between the numeric value and the scale factor when units are present. Is generally specified to be '' or ' '; use the latter if you prefer a space between the number and the units. Generally using ' ' makes numbers easier to read, particularly with complex units, and using '' is easier to parse. You could also use a Unicode non-breaking space ' '. For your convenience, you can access a non-breaking space using Quantity.non_breaking_space.

- **strip_radix** (*bool*) – When rendering, strip the radix (decimal point) from numbers even if they can then be mistaken for integers. By default this is True.

- **unity_sf** (*str*) – The output scale factor for unity, generally '' or '_'. The default is '', but use '_' if you want there to be no ambiguity between units and scale factors. For example, 0.3 would be rendered as '300m', and 300 m would be rendered as '300_m'.

Raises **KeyError** – unknown preference.

Example:

```
>>> mu0 = Quantity('mu0')
>>> print(mu0)
1.2566 uH/m

>>> Quantity.set_prefs(prec=6, map_sf={'u': 'μ'})
>>> print(mu0)
1.256637 μH/m

>>> Quantity.set_prefs(prec=None, map_sf=None)
>>> print(mu0)
1.2566 uH/m
```

Unit Conversion

class quantiphy.**UnitConversion**(*to_units*, *from_units*, *slope=1*, *intercept=0*)

Creates a unit converter. Just the creation of the converter is sufficient to make it available to `Quantity` (the `UnitConversion` object itself is normally discarded). Once created, it is automatically employed by `Quantity` when a conversion is requested with the given units. A forward conversion is performed if the from and to units match, and a reversion conversion is performed if they are swapped.

Parameters

- **to_units** (*string or list of strings*) – A collection of units. If given as a single string it is split.
- **from_units** (*string or list of strings*) – A collection of units. If given as a single string it is split.
- **slope** (*float*) – Scale factor for conversion.
- **intercept** (*float*) – Conversion offset.

Forward Conversion: The following conversion is applied if the given units are among the *from_units* and the desired units are among the *to_units*:

new_value = *given_value* * *slope* + *intercept*

Reverse Conversion: The following conversion is applied if the given units are among the *to_units* and the desired units are among the *from_units*:

new_value = (*given_value* - *intercept*)/*slope*

Example:

```
>>> from quantiphy import Quantity, UnitConversion
>>> UnitConversion('m', 'pc parsec', 3.0857e16)
<...>
```

This establishes a conversion between meters (m) and parsecs (parsec, pc) that can go both ways:

```
>>> d_sol = Quantity('5 µpc', scale='m')
>>> print(d_sol)
154.28 Gm

>>> d_ac = Quantity(1.339848, units='pc')
>>> print(d_ac.render(scale='m'))
41.344e15 m
```

Constants and Unit Systems

quantiphy.**add_constant**(*value*, *alias=None*, *unit_systems=None*)

Saves a quantity in such a way that it can later be recalled by name when creating new quantities.

Parameters

- **value** (*quantity*) – The value of the constant. Must be a quantity or a string that can be directly converted to a quantity.
- **alias** (*str*) – An alias for the constant. Can be used to access the constant from as an alternative to the name given in the value, which itself is optional. If the value has a name, specifying this name is optional. If both are given, the constant is accessible using either name.
- **unit_systems** (*list or str*) – Name or names of the unit systems to which the constant should be added. If given as a string, string will be split at white space to create the list. If a constant is associated with a unit system, it is only available when that unit system is active. You need not limit yourself to the predefined 'mks' and 'cgs' unit systems. Giving a name creates the corresponding unit system if it does not already exist. If *unit_systems* is not given, the constant is not associated with a unit system, meaning that it is always available regardless of which unit system is active.

Raises

- **ValueError** – *value* must be an instance of `Quantity` or it must be a string that can be converted to a quantity.
- **NameError** – *alias* was not specified and no name was available from *value*.

The constant is saved under *name* if given, and under the name contained within *value* if available. It is not necessary to supply both names, one is sufficient.

Example:

```
>>> from quantiphy import Quantity, add_constant
>>> add_constant('f_hy = 1420.405751786 MHz -- Frequency of hydrogen line')
>>> print(Quantity('f_hy').render(show_label='f'))
f_hy = 1.4204 GHz -- Frequency of hydrogen line
```

`quantiphy.`**`set_unit_system`**(*unit_system*)

Activates a unit system.

The default unit system is 'mks'. Calling this function changes the active unit system to the one with the specified name. Only constants associated with the active unit system or not associated with a unit system are available for use.

Parameters **`unit_system`** (*str*) – Name of the desired unit system.

A *KeyError* is raised if *unit_system* does not correspond to a known unit system.

Example:

```
>>> from quantiphy import Quantity, set_unit_system
>>> set_unit_system('cgs')
>>> print(Quantity('h').render(show_label='f'))
h = 6.6261e-27 erg-s -- Plank's constant

>>> set_unit_system('mks')
>>> print(Quantity('h').render(show_label='f'))
h = 662.61e-36 J-s -- Plank's constant
```

Examples

Motivating Example

QuantiPhy is a light-weight package that allows numbers to be combined with units into physical quantities. Physical quantities are very commonly encountered when working with real-world systems when numbers are involved. And when encountered, the numbers often use SI scale factors to make them easier to read and write. Surprisingly, most computer languages do not support numbers in these forms, meaning that when working with physical quantities, one often has to choose between using a form that is easy for computers to read or one that is easy for humans to read. For example, consider this table of critical frequencies needed in jitter tolerance measurements in optical communication:

```
>>> table1 = """
...     SDH     | Rate          | f1      | f2       | f3      | f4
...     --------+---------------+---------+----------+---------+--------
...     STM-1   | 155.52 Mb/s   | 500 Hz  | 6.5 kHz  | 65 kHz  | 1.3 MHz
...     STM-4   | 622.08 Mb/s   | 1 kHz   | 25 kHz   | 250 kHz | 5 MHz
...     STM-16  | 2.48832 Gb/s  | 5 kHz   | 100 kHz  | 1 MHz   | 20 MHz
...     STM-64  | 9.95328 Gb/s  | 20 kHz  | 400 kHz  | 4 MHz   | 80 MHz
...     STM-256 | 39.81312 Gb/s | 80 kHz  | 1.92 MHz | 16 MHz  | 320 MHz
... """
```

This table was formatted to be easily read by humans. If it were formatted for computers, the numbers would be given without units and in exponential notation because they have dramatically different sizes. For example, it might look like this:

```
>>> table2 = """
...     SDH     | Rate (b/s)     | f1 (Hz) | f2 (Hz)   | f3 (Hz) | f4 (Hz)
...     --------+----------------+---------+-----------+---------+--------
...     STM-1   | 1.5552e8       | 5e2     | 6.5e3     | 6.5e3   | 1.3e6
...     STM-4   | 6.2208e8       | 1e3     | 2.5e3     | 2.5e5   | 5e6
...     STM-16  | 2.48832e9      | 5e3     | 1e5       | 1e6     | 2e7
...     STM-64  | 9.95328e9      | 2e4     | 4e5       | 4e6     | 8e7
...     STM-256 | 3.981312e10    | 8e4     | 1.92e6    | 1.6e7   | 3.20e8
... """
```

This contains the same information, but it is much harder for humans to read and interpret. Often the compromise of partially scaling the numbers can be used to make the table easier to interpret:

```
>>> table3 = """
...     SDH     | Rate (Mb/s)    | f1 (kHz)| f2 (kHz)  | f3 (kHz)| f4 (MHz)
...     --------+----------------+---------+-----------+---------+---------
...     STM-1   | 155.52         | 0.5     | 6.5       | 65      | 1.3
...     STM-4   | 622.08         | 1       | 2.5       | 250     | 5
...     STM-16  | 2488.32        | 5       | 100       | 1000    | 20
...     STM-64  | 9953.28        | 20      | 400       | 4000    | 80
...     STM-256 | 39813.12       | 80      | 1920      | 16000   | 320
... """
```

This looks cleaner, but it is still involves some effort to interpret because the values are distant from their corresponding scaling and units, because the large and small values are oddly scaled (0.5 kHz is more naturally given as 500Hz and 39813 MHz is more naturally given as 39.8 GHz), and because each column may have a different scaling factor. While these might seem like minor inconveniences on this table, they can become quite annoying as tables become larger or more numerous. Fundamentally the issue is that the eyes are naturally drawn to the number, but the numbers are not complete, and so the eyes need to hunt further. This problem exists with both tables and graphs. The scaling and units for the numbers may be found in the column headings, the axes, the labels, the title, the caption, or in the body of the text. The sheer number of places to look can dramatically slow the interpretation of the data. This problem does not exist in the first table where each number is complete as it includes both its scaling and its units. The eye gets the full picture on the first glance.

This last version of the table represents a very common mistake people make when presenting data. They feel that adding units and scale factors to each number adds clutter and wastes space and so removes them from the data and places them somewhere else. Doing so results in a data that perhaps is visually cleaner but is harder for the reader to interpret. All these tables contain the same information, but in the second two tables the readability has been traded off in order to make the data easier to read into a computer because in most languages there is no easy way to read numbers that have either units or scale factors.

QuanitiPhy makes it easy to read and generate numbers with units and scale factors so you do not have to choose between human and computer readability. For example, the above tables could be read with the following code (it must be tweaked somewhat to handle tables 2 and 3):

```
>>> from quantiphy import Quantity

>>> # parse the table
>>> sdh = []
>>> lines = table1.strip().split('\n')
>>> for line in lines[2:]:
...     fields = line.split('|')
...     name = fields[0].strip()
...     rate = Quantity(fields[1])
```

```
...     critical_freqs = [Quantity(f) for f in fields[2:]]
...     sdh.append((name, rate, critical_freqs))

>>> # print the table in a form suitable for humans
>>> for name, rate, freqs in sdh:
...     print('{:8s}: {:12s} {:9s} {:9s} {:9s} {}'.format(name, rate, *freqs))
STM-1   : 155.52 Mb/s  500 Hz     6.5 kHz    65 kHz     1.3 MHz
STM-4   : 622.08 Mb/s  1 kHz      25 kHz     250 kHz    5 MHz
STM-16  : 2.4883 Gb/s  5 kHz      100 kHz    1 MHz      20 MHz
STM-64  : 9.9533 Gb/s  20 kHz     400 kHz    4 MHz      80 MHz
STM-256 : 39.813 Gb/s  80 kHz     1.92 MHz   16 MHz     320 MHz

>>> # print the table in a form suitable for machines
>>> for name, rate, freqs in sdh:
...     print('{:8s}: {:.4e} {:.4e} {:.4e} {:.4e} {:.4e}'.format(name, rate, *(1*f␣
↪for f in freqs)))
STM-1   : 1.5552e+08 5.0000e+02 6.5000e+03 6.5000e+04 1.3000e+06
STM-4   : 6.2208e+08 1.0000e+03 2.5000e+04 2.5000e+05 5.0000e+06
STM-16  : 2.4883e+09 5.0000e+03 1.0000e+05 1.0000e+06 2.0000e+07
STM-64  : 9.9533e+09 2.0000e+04 4.0000e+05 4.0000e+06 8.0000e+07
STM-256 : 3.9813e+10 8.0000e+04 1.9200e+06 1.6000e+07 3.2000e+08
```

The code first reads the data and then produces two outputs. The first output shows that quantities can be displayed in easily readable forms with their units and the second output shows that the values are easily accessible for computation (the use of `1*f` is not necessary to be able to see the results in exponential notation, rather it is there to demonstrate that it is easy to do calculations on Quantities).

quantiphy.Quantity is used to convert a number string, such as '155.52 Mb/s' into an internal representation that includes the value and the units: 155.52e6 and 'b/s'. The scaling factor is properly interpreted. Once a value is converted to a *Quantity*, it can be treated just like a normal *float*. The main difference occurs when it is time to convert it back to a string. When doing so, the scale factor and units are included by default.

DRAM Prices

Here is a table that was found on the Internet that gives the number of bits of dynamic RAM a dollar would purchase over time:

```
>>> bits_per_dollar = '''
...     1973 490
...     1978 2780
...     1983 16400
...     1988 91800
...     1993 368000
...     1998 4900000
...     2003 26300000
...     2008 143000000
...     2013 833000000
...     2018 5000000000
... '''
```

It is pretty easy to read in the early years, but by the turn of the millennium you have to start counting the zeros by hand to understand the number. And are those bits or bytes? Reformatting with *QuantiPhy* makes it much more readable:

```
>>> for line in bits_per_dollar.strip().split('\n'):
...     year, bits = line.split()
...     bits = Quantity(bits, 'b')
```

```
...     print(f'{year}    {bits:7q}    {bits:qB}')
1973    490 b       61.25 B
1978    2.78 kb     347.5 B
1983    16.4 kb     2.05 kB
1988    91.8 kb     11.475 kB
1993    368 kb      46 kB
1998    4.9 Mb      612.5 kB
2003    26.3 Mb     3.2875 MB
2008    143 Mb      17.875 MB
2013    833 Mb      104.12 MB
2018    5 Gb        625 MB
```

Notice that *bits* was printed twice. The first time the formatting code included a width specification, but in the second the desired unit of measure was specified (*B*), which caused the underlying value to be converted from bits to bytes.

It is important to recognize that *QuantiPhy* is using decimal rather than binary scale factors. So 5 GB is 5 gigabyte and not 5 gibibyte. In otherwords 5 GB represents $5{\times}10^9$ B and not $5{\times}2^{30}$ B.

Thermal Voltage Example

In this example, quantities are used to represent all of the values used to compute the thermal voltage: *Vt = kT/q*. It is not terribly useful, but does demonstrate several of the features of *QuantiPhy*.

```
>>> from quantiphy import Quantity
>>> with Quantity.prefs(
...     show_label = 'f',
...     label_fmt = '{n} = {v}',
...     label_fmt_full = '{V:<18}  # {d}',
... ):
...     T = Quantity(300, 'T K ambient temperature')
...     k = Quantity('k')
...     q = Quantity('q')
...     Vt = Quantity(k*T/q, 'Vt V thermal voltage')
...     print(T, k, q, Vt, sep='\n')
T = 300 K           # ambient temperature
k = 13.806e-24 J/K  # Boltzmann's constant
q = 160.22e-21 C    # elementary charge
Vt = 25.852 mV      # thermal voltage
```

The first part of this example imports *quantiphy.Quantity* and sets the *show_label* and *label_fmt* preferences to display both the value and the description by default. *label_fmt* is given as a tuple of two strings, the first will be used when the description is present, the second is used when it is not. In the first string, the `{V:<16}` is replaced by the expansion of the second string, left justified with a field width of 16, and the `{d}` is replaced by the description. On the second string the `{n}` is replaced by the *name* and `{v}` is replaced by the value (numeric value and units).

The second part defines four quantities. The first is given in a very specific way to avoid the ambiguity between units and scale factors. In this case, the temperature is given in Kelvin (K), and normally if the temperature were given as the string '300 K', the units would be confused for the scale factor. As mentioned in *Ambiguity of Scale Factors and Units* the 'K' would be treated as a scale factor unless you took explicit steps. In this case, this issue is circumvented by specifying the units in the *model* along with the name and description. The *model* is also used when creating *Vt* to specify the name, units, and description.

The last part simply prints the four values. The *show_label* preference is set so that names and descriptions are printed along with the values. In this case, since all the quantities have descriptions, *label_fmt_full* is used to format the output.

Unicode Text Example

In this example *QuantiPhy* formats quantities to be embedded in text. To make the text as clean as possible, *QuantiPhy* is configured to use Unicode scale factors and the Unicode non-breaking space as the spacer. The non-breaking space prevents units from being placed on a separate line from their number, making the quantity easier to read.

```
>>> from quantiphy import Quantity
>>> import textwrap

>>> Quantity.set_prefs(
...     map_sf = Quantity.map_sf_to_sci_notation,
...     spacer = Quantity.non_breaking_space
... )

>>> constants = [
...     Quantity('h'),
...     Quantity('hbar'),
...     Quantity('k'),
...     Quantity('q'),
...     Quantity('c'),
...     Quantity('0C'),
...     Quantity('eps0'),
...     Quantity('mu0'),
... ]

>>> # generate some sentences that contain quantities
>>> sentences = [f'{q.desc.capitalize()} is {q}.' for q in constants]

>>> # combine the sentences into a left justified paragraph
>>> print(textwrap.fill('  '.join(sentences)))
Plank's constant is 662.61×10^-36 J-s.  Reduced plank's constant is
105.46×10^-36 J-s.  Boltzmann's constant is 13.806×10^-24 J/K.
Elementary charge is 160.22×10^-21 C.  Speed of light is 299.79 Mm/s.
Zero degrees celsius in kelvin is 273.15 K.  Permittivity of free
space is 8.8542 pF/m.  Permeability of free space is 1.2566 μH/m.
```

When rendered in your browser with a variable width font, the result looks like this:

> Plank's constant is 662.61×10^{-36} J-s. Reduced plank's constant is 105.46×10^{-36} J-s. Boltzmann's constant is 13.806×10^{-24} J/K. Elementary charge is 160.22×10^{-21} C. Speed of light is 299.79 Mm/s. Zero degrees celsius in kelvin is 273.15 K. Permittivity of free space is 8.8542 pF/m. Permeability of free space is 1.2566 μH/m.

Timeit Example

A Python module that benefits from *QuantiPhy* is *timeit*, a package in the standard library that runs a code snippet a number of times and prints the elapsed time for the test. However, from a usability perspective it has several issues. First, it prints out the elapsed time of all the repetitions rather than dividing the elapsed time by the number of repetitions and reporting the average time per operation. So it can quickly allow you to compare the relative speed of various operations, but it does not directly give you a sense of the time required in absolute terms. Second, it does not label its output, so it is not clear what is being displayed. Here is an example where *timeit* has been fortified with *QuantiPhy* to make the output more readable. To make it more interesting, the timing results are run on *QuantiPhy* itself. The results give you a feel for how much slower *QuantiPhy* is to both convert strings to quantities and quantities to strings compared into the built-in float class.

```
#!/usr/bin/env python3
from timeit import timeit
from random import random, randint
from quantiphy import Quantity

# preferences
trials = 100_000
Quantity.set_prefs(
    prec = 2,
    show_label = True,
    label_fmt = '{n:>40}: {v}',
    map_sf = Quantity.map_sf_to_greek
)

# build the raw data, arrays of random numbers
s_numbers = []
s_quantities = []
numbers = []
quantities = []
for i in range(trials):
    mantissa = 20*random()-10
    exponent = randint(-35, 35)
    number = '%0.25fe%s' % (mantissa, exponent)
    quantity = number + ' Hz'
    s_numbers.append(number)
    s_quantities.append(quantity)
    numbers.append(float(number))
    quantities.append(Quantity(number, 'Hz'))

# define testcases
testcases = [
    '[float(v) for v in s_numbers]',
    '[Quantity(v) for v in s_quantities]',
    '[str(v) for v in numbers]',
    '[str(v) for v in quantities]',
]

# run testcases and print results
print(f'For {Quantity(trials)} values ...')
for case in testcases:
    elapsed = timeit(case, number=1, globals=globals())
    result = Quantity(elapsed/trials, units='s/op', name=case)
    print(result)
```

The results are:

```
For 100k iterations ...
            [float(v) for v in s_numbers]: 638 ns/op
     [Quantity(v) for v in s_quantities]: 15.3 µs/op
               [str(v) for v in numbers]: 1.03 µs/op
            [str(v) for v in quantities]: 28.1 µs/op
```

You can see that *QuantiPhy* is considerably slower than the float class, which you should be aware of if you are processing large quantities of numbers.

Contrast this with the normal output from *timeit*:

```
0.05213119700783864
1.574107409993303
0.10471829099697061
2.3749650190002285
```

The essential information is there, but it takes longer to make sense of it.

Disk Usage Example

Here is a simple example that uses *QuantiPhy* to clean up the output from the Linux disk usage utility. It runs the *du* command, which prints out the disk usage of files and directories. The results from *du* are gathered and then sorted by size and then the size and name of each item is printed.

Quantity is used to scale the filesize reported by *du* from KB to B. Then the list of files is sorted by size. Here we are exploiting the fact that quantities act like floats, and so the sorting can be done with no extra effort. Finally, the ability to render to a number with a scale factor and units is used when presenting the results.

```
#!/usr/bin/env python3
# runs du and sorts the output while suppressing any error messages from du

from quantiphy import Quantity
from inform import display, fatal, os_error
from shlib import Run
import sys

try:
    du = Run(['du'] + sys.argv[1:], modes='WEO1')

    files = []
    for line in du.stdout.split('\n'):
        if line:
            size, filename = line.split('\t', 1)
            files += [(Quantity(size, scale=(1000, 'B')), filename)]

    files.sort(key=lambda x: x[0])

    for each in files:
        display('{:>7.2s}  {}'.format(*each))

except OSError as err:
    fatal(os_error(err))
except KeyboardInterrupt:
    display('dus: killed by user.')
```

And here is an example of the programs output:

```
 460 kB  quantiphy/examples/delta-sigma
 464 kB  quantiphy/examples
1.54 kB  quantiphy/doc
3.48 MB  quantiphy
```

MatPlotLib Example

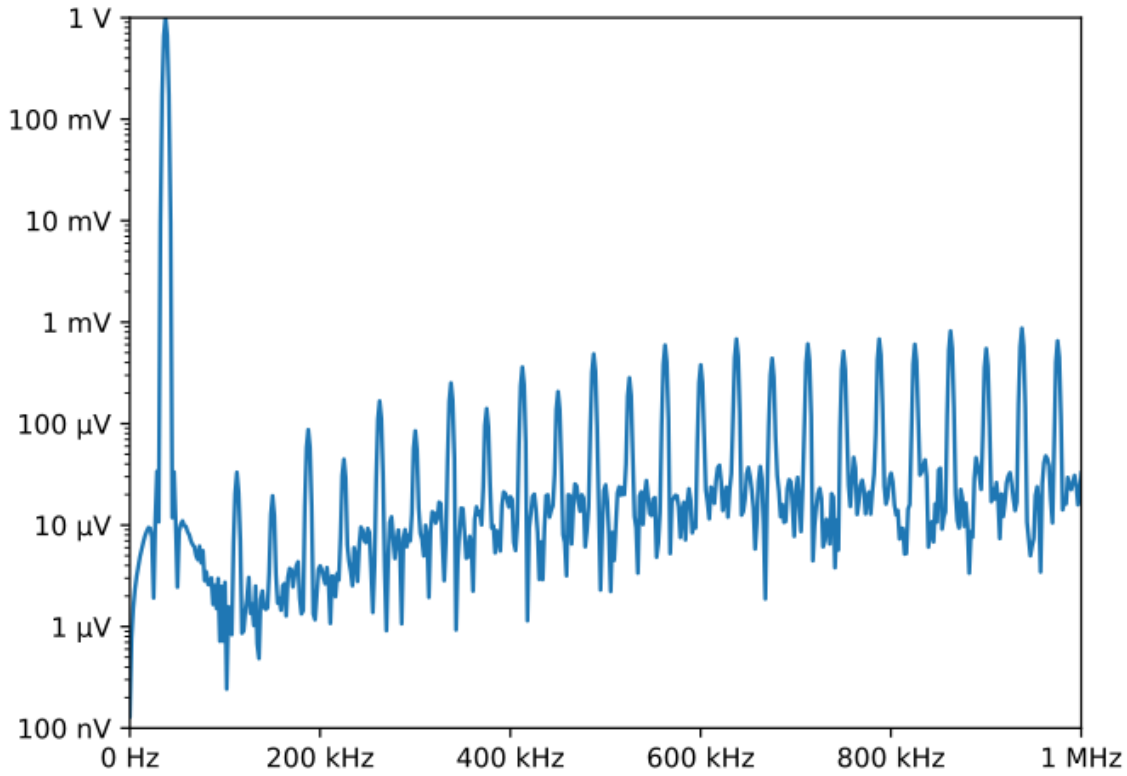
In this example *QuantiPhy* is used to create easy to read axis labels in MatPlotLib. It uses NumPy to do a spectral analysis of a signal and then produces an SVG version of the results using MatPlotLib.

```
#!/usr/bin/env python3

import numpy as np
from numpy.fft import fft, fftfreq, fftshift
import matplotlib as mpl
mpl.use('SVG')
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as pl
from quantiphy import Quantity
Quantity.set_prefs(map_sf=Quantity.map_sf_to_sci_notation)

# read the data from delta-sigma.smpl
data = np.fromfile('delta-sigma.smpl', sep=' ')
time, wave = data.reshape((2, len(data)//2), order='F')

# print out basic information about the data
timestep = Quantity(time[1] - time[0], name='Time step', units='s')
nonperiodicity = Quantity(wave[-1] - wave[0], name='Nonperiodicity', units='V')
points = Quantity(len(time), name='Time points')
period = Quantity(timestep * len(time), name='Period', units='s')
freq_res = Quantity(1/period, name='Frequency resolution', units='Hz')
with Quantity.prefs(show_label=True, prec=2):
    print(timestep, nonperiodicity, points, period, freq_res, sep='\n')

# create the window
window = np.kaiser(len(time), 11)/0.37
    # beta=11 corresponds to alpha=3.5 (beta = pi*alpha)
    # the processing gain with alpha=3.5 is 0.37
windowed = window*wave

# transform the data into the frequency domain
spectrum = 2*fftshift(fft(windowed))/len(time)
freq = fftshift(fftfreq(len(wave), timestep))

# define the axis formatting routines
freq_formatter = FuncFormatter(lambda v, p: str(Quantity(v, 'Hz')))
volt_formatter = FuncFormatter(lambda v, p: str(Quantity(v, 'V')))

# generate graphs of the resulting spectrum
fig = pl.figure()
ax = fig.add_subplot(111)
ax.plot(freq, np.absolute(spectrum))
ax.set_yscale('log')
ax.xaxis.set_major_formatter(freq_formatter)
ax.yaxis.set_major_formatter(volt_formatter)
pl.savefig('spectrum.svg')
ax.set_xlim((0, 1e6))
ax.set_ylim((1e-7, 1))
pl.savefig('spectrum-zoomed.svg')
```

This script produces the following textual output:

```
Time step = 20 ns
Nonperiodicity = 2.3 pV
Time points = 28k
Period = 560 µs
Frequency resolution = 1.79 kHz
```

And the following is one of the two graphs produced:



Notice the axis labels in the generated graph. Use of *QuantiPhy* makes the widely scaled units compact and easy to read.

MatPlotLib provides the EngFormatter that you can use as an alternative to *QuantiPhy* for formatting your axes with SI scale factors, which also provides the *format_eng* function for converting floats to strings formatted with SI scale factors and units. So if your needs are limited, as they are in this example, that is generally a good way to go. One aspect of *QuantiPhi* that you might prefer is they way it handles very large or very small numbers. As the numbers get either very large or very small *EngFormatter* starts by using unfamiliar scale factors (*YZPEzy*) and then reverts to e-notation. *QuantiPhy* allows you to control whether to use unfamiliar scale factors but does not use them by default. It also can be configured to revert to engineering scientific notation (ex: $13.806{\times}10^{24}$ J/K) when no scale factors are appropriate. Though not necessary for this example, that was done above with the line:

```
Quantity.set_prefs(map_sf=Quantity.map_sf_to_sci_notation)
```

Releases

1.0 (2016-11-26):

- Initial production release.

1.1 (2016-11-27):

- Added *known_units* preference.

- Added *get_preference* class method.

1.2 (2017-02-24):

- allow digits after decimal point to be optional
- support underscores in numbers
- allow options to be monkey-patched on to Quantity objects
- add strip_dp option
- fix some issues in full precision mode
- ranamed some options, arguments and methods

1.3 (2017-03-19):

- reworked constants
- added unit systems for physical constants

2.0 (2017-07-15): This is a 'coming of age' release where the emphasis shifts from finding the right interface to providing an interface that is stable over time. This release includes the first formal documentation and a number of new features and refinements to the API.

- created formal documentation
- enhanced label_fmt to accept {V}
- allow quantity to be passed as value to Quantity
- replaced Quantity.add_to_namespace with Quantity.extract
- raise NameError rather than Assertion for unknown preferences
- added Quantity.all_from_conv_fmt() and Quantity.all_from_si_fmt()
- change assign_rec to support more formats
- changed Constant() to add_constant()
- changed the way preferences are implemented
- changed name of preference methods: set_preferences -> set_prefs, get_preference -> get_pref
- added Quantity.prefs() (preferences context manager)
- split label_fmt preference into two: label_fmt and label_fmt_full
- added show_desc preference
- allow show_label to be either 'a' or 'f' as well True or False
- renamed strip_dp option to strip_radix
- added number_fmt option

2.1 (2017-07-30): The primary focus of this release was on improving the documentation, though there are a few small feature enhancements.

- added support for SI standard composite units
- added support for non-breaking space as spacer
- removed constraint in extract() that names must be identifiers

Index