Quantity-Value Documentation

Release 0.3.0.dev0

Measurement Standards Laboratory of New Zealand

Aug 09, 2023

CONTENTS

CHAPTER

ONE

LICENSE

CHAPTER

TWO

INTRODUCTION

This package supports the representation of physical quantities as a value paired with a unit, for example 10.5 kg. It is possible to declare quantities like `m = qvalue(10.5,kg)` and then use `m` in mathematical expressions. The rules governing calculations with quantities are handled by the software.

We intend the package to explore methods of implementing quantity-correctness in calculations with physical quantities, but it is still in the early stages of development.

2.1 Background

The correct way to express a physical quantity, such as the mass 940 kg, is as a measure (a number) paired with a unit, where the unit is associated with the same kind of quantity as is being measured (the unit is in fact the name of the measurement scale used). Writing the measure alone is not enough, because information about what can be done with the measure is lost. The rules of calculus when quantities are involved are not the same as pure numbers. For instance, only quantities of the same kind may be added; so, 940 kg + 60 kg is permitted, and evaluates to 1000 kg, but 940 kg + 60 cannot be evaluated.

When digital systems handle data representing physical quantities, one might expect software support for physical quantities to be used. However, this is rarely the case. Why this should be so is not clear, perhaps no need is perceived; but if so, that would be wishful thinking. Famous failures to handle physical data correctly include the loss of NASA's Mars Climate Observer, in 1999, and the so-called Gimli Glider incident, in 1983, when a commercial Air Canada flight ran out of fuel. Support is needed now more than ever, because there is a huge increase in the amount of measurement data being generated and processed automatically by digital systems.

From early days of general-purpose computing in the 1970s to the present, many suggestions have been made about how to support quantities, but no general solution has emerged. It appears that the challenge of encoding the semantics of physical quantity data in software is harder than it appears. This project is trying to use contextual information about a problem to tailor support to the context. This is similar to what happens in scientific writing, where careful attention must be given to a description of terms when quantities are involved. Without this, a text becomes obscure and only a reader familiar with the missing contextual information can make sense of it. Similar principles should apply to software representations of quantities.

Our work addresses the difficulties encountered in the past when units are encoded by dimensional exponents in a conventional basis. When quantity calculus is then used to track quantities represented by dimensional exponents, there are inevitably problems of ambiguity and difficulties of expression. For instance, fuel consumption is conventionally expressed in L/(100 km) but has the dimensions of area (in the SI), whereas rainfall, when measured as a volume divided by a cross-sectional area, is conventionally simplified and reported as a length. There are also many so-called 'dimensionless' quantities which are effectively unit-less, but certainly should not be considered pure numbers. And then there is the problem of disambiguating quantities like torque, work and energy, which have the SI dimensions M^2LT^{-2}, because the choice of base quantities is inadequate.

2.1.1 Kind of quantity, individual quantity and quantity value

A distinction is made between a *kind of quantity* and an *individual quantity*. Kind of quantity is the more general concept, for example: *length*, *mass*, etc, [VIM_S1.2]. Whereas, an individual quantity is specific, for example: the mass of a particular car, m_{car}, 940 kilograms, etc. Also, when an individual quantity is presented as a measure paired with a unit, it may be called a *quantity value* [VIM_S1.19]. When the intended meaning is clear in the context, we will simply use *quantity* for any of: 'kind of quantity', 'individual quantity' or 'quantity value'.

2.1.2 Dimensions and dimensional exponents

Dimensions were introduced by Joseph Fourier, in 1822, as a way to calculate the scale factor to adjust a measure when units are changed. Dimensions are usually expressed as a product of variables (one for each base quantity dimension) that may be exponentiated. The symbols chosen recall the kind of quantity on which the measure depends (so they often *appear* to be associated with the kind of quantity, when really they are associated with the particular scale used). For example, the dimensions for a measure of speed in the SI are LT^{-1}, where L is the variable (dimension) associated with length and T the variable (dimension) associated with time. The dimensional exponent -1 applied to T means that L is divided by T. So, to transform a speed of 50 km/h into metres per second, 50 will be multiplied by 1000/3600, giving a speed of approximately 13.89 m/s.

2.1.3 Quantity calculus

The rules of quantity calculus are usually presented as arithmetic operations applied to dimensions. Such as: the dimensions of a product of quantity values is obtained as the products of the corresponding dimensions of each factor (so the exponents of the same dimensional variable are added). For example, the distance covered by a body, initially at rest, moving with a constant acceleration, a, for a time, t, is $s = \frac{1}{2}at^2$. In the SI, the dimensions for a are LT^{-2} and the dimensions for an elapsed time squared are T^2. So, on the right-hand side of the equation, the product of dimensions, L, matches the dimension of the distance, s. (When dimensions balance on either side of an equation it is called *dimensional homogeneity* and is a necessary condition for equality. Homogeneity ensures that the equation remains true even when the measurement units change.)

2.1.4 Dimensional analysis

Dimensional analysis is an analytical technique that can be used to study relationships among quantities subject to physical laws. It is based on the idea that physical laws do not depend on the units in which quantities are measured.

The seven familiar SI base units and the convention of representing other units as a product of base-unit symbols, possibly exponentiated, is an application of dimensional analysis [SI_brochure]. However, it is important to remember that the choice of SI dimensions is conventional. There is still freedom to choose the most convenient set of dimensions for a particular problem. It is also useful to remember that SI symbols are really just names for scales. The process of generating a name (by multiplying and dividing the names of base units) does not guarantee the existence of a meaningful physical quantity to be measured.

CHAPTER

THREE

EXAMPLES

3.1 Simple kinematics

When using the package, the first task is to select a set of base quantities.

For instance, the base quantities distance and duration (dimensions, LT) may be used for a straight-line kinematics problems. Other kinds of quantity are then declared in terms of this basis. For example, speed is the time required to cover a distance.

```
from QV import *

quantity = Context( ("Length","L"),("Time","T") )
quantity.declare('Speed','V','Length/Time')
```

Here, `context` maintains one-to-one relationships between the names, and short symbols, of kinds of quantities and the signatures associated with measurements of them. So, after the declaration of speed, `context` does not allow any other quantity to be declared with the same signature.

Units are defined in relation to kinds of quantity. In this case, we might write

```
SI =  UnitRegister("SI",quantity)

metre = SI.unit( RatioScale(quantity.Length,'metre','m') )
second = SI.unit( RatioScale(quantity.Time,'second','s') )
metre_per_second = SI.unit( RatioScale(quantity.Speed,'metre_per_second','m/s') )
```

Here, the `SI` object keeps a register of units, each associated with the measurement of a kind of quantity and hence to the signature of that quantity. The first `unit` declaration for a quantity creates a reference unit within the register; other units of the same kind of quantity can also be registered, but they must be related to the reference unit by a conversion factor (see below, where a related unit, L/(100 km), is created for fuel consumption.)

Quantity values may be defined with the function `qvalue()` and used in calculations. For instance,

```
d = qvalue(0.5,metre)
t = qvalue(1.0,second)
print( "average speed =", qresult(d/t) )

v0 = qvalue(5.2,metre_per_second)
x0 = qvalue(0.3,metre)
print( "displacement =", x0 + v0*t )
```

The output is

```
average speed = 0.5 m/s
displacement = 5.5 m
```

An interesting implementation detail is apparent here. The function `qresult()` is applied to `d/t` to resolve the units, but it is not used in the calculation of `x0 + v0*t`. The reason is that individual multiplications or divisions are often just intermediate steps in a calculation. So, QV will not try to resolve the kind of quantity of an operation until signalled to do so. However, addition and subtraction of different kinds of quantity is not allowed. So, the arguments in the sum `x0 + v0*t` must be checked, and this requires QV to resolve the units of `v0*t`.

3.2 Fuel consumption

This package facilitates the use of *ad hoc* units. For example, fuel consumption is typically stated in units of litres per 100 km. This can be handled as follows[1]

```
from fractions import Fraction

quantity = Context( ("Distance","L"), ("Volume","V") )
FuelConsumption = quantity.declare( 'FuelConsumption','FC','Volume/Distance' )

ureg =  UnitRegister("ureg",quantity)

# Reference units
kilometre = ureg.unit( RatioScale(quantity['Distance'],'kilometre','km') )
litre = ureg.unit( RatioScale(quantity['Volume'],'litre','L') )
litres_per_km = ureg.unit( RatioScale(quantity['FuelConsumption'],'litres_per_km','L/
↪km' ) )

litres_per_100_km = ureg.unit(
    proportional_unit(
        litres_per_km,
        'litres_per_100_km','L/(100 km)',
        Fraction(1,100)
    )
)
```

Calculations proceed as might be expected

```
distance = qvalue(25.6,kilometre)
fuel = qvalue(2.2,litre)

consumes = fuel/distance
print( "average consumption =", qresult( consumes, litres_per_100_km ) )

distance = qvalue(155,kilometre)
print( 'fuel required =', qresult( consumes * distance ) )
```

which gives the following results[2].

```
average consumption = 8.59375 L/(100 km)
fuel required = 13.3203125 L
```

It is interesting that QV can treat distance and volume as quite distinct quantities, although they share the dimension of length in the SI[3].

[1] The distance reference unit could have been chosen as 100 km, instead of 1 km, but it seems more natural to proceed as shown. The reference unit for consumption, `litres_per_km`, is determined by the reference units for volume and distance. The related unit of `litres_per_100_km` must be introduced with an appropriate scale factor.

[2] The argument `litres_per_100_km` is passed to `qresult()` to obtain results in the required unit. The default would be the reference unit declared for the kind of quantity (`litres_per_km` in this case).

[3] Reduced to SI base units, the consumption is about $8.6 \times 10^{-8}\ m^2$. This area, multiplied by the distance travelled, is the volume of fuel required.

3.3 Electrical quantities

Electrical measurements involve particular quantities, and associated units. We can use base quantities V, I and T, for potential difference, current and duration, respectively. Then additional quantities of interest include: resistance, capacitance, inductance, energy, power and angular frequency. The context can be configured, as follows

```
quantity = Context( ("Current","I"),("Voltage","V"),("Time","T") )

quantity.declare('Resistance','R','Voltage/Current')
quantity.declare('Capacitance','C','I*T/V')
quantity.declare('Inductance','L','V*T/I')
quantity.declare('Angular_frequency','F','1/T')
quantity.declare('Power','P','V*I')
quantity.declare('Energy','E','P*T')
```

Suitable units are:

```
ureg =  UnitRegister("Reg",quantity)

volt = ureg.unit( RatioScale(quantity.Voltage,'volt','V') )
second = ureg.unit( RatioScale(quantity.Time,'second','s') )
ampere = ureg.unit( RatioScale(quantity.Current,'ampere','A') )
ohm = ureg.unit( RatioScale(quantity.Resistance,'Ohm','Ohm') )
henry = ureg.unit( RatioScale(quantity.Inductance,'henry','H') )
rad_per_s = ureg.unit( RatioScale(quantity.Angular_frequency,'radian_per_second','rad/
↪s') )
watt = ureg.unit( RatioScale(quantity.Power,'watt','W') )
joule = ureg.unit( RatioScale(quantity.Energy,'joule','J') )
```

Calculations are then straightforward. For example,

```
from math import pi

v1 = qvalue(0.5,volt)
i1 = qvalue(1.E-3,ampere)
l1 = qvalue(0.3E-3,henry)
w1 = qvalue(2*pi*2.3E3,rad_per_s)

r1 = v1/i1

print( "resistance =", qresult(r1) )
print( "reactance =", qresult(w1*l1) )
print( "energy =", qresult(0.5*l1*i1*i1) )
print( "power =", qresult(v1*i1) )

r2 = qvalue(2.48E3,ohm)
print(  "parallel resistance =",  qresult( (r1*r2)/(r1 + r2) ) )
```

Which produces

```
resistance = 500.0 Ohm
reactance = 4.33539786195 Ohm
energy = 1.5e-10 J
power = 0.0005 W
parallel resistance = 416.10738255 Ohm
```

3.4 Ratios

Ratios of the same quantities arise frequently in calculations. These ratios are often described as *dimensionless*, but they are not plain numbers and the quantities involved should not be ignored.

Dimensionless ratios can retain quantity information if defined using the function `qratio`.

For example, continuing the electrical case above (where `r1` and `r2` were evaluated), a resistor network may be used to scale down a voltage by some fraction (often called a potential, or resistive, divider). The resistance ratio can be defined as a dimensionless quantity in this way

```
quantity.declare( 'Resistance_ratio','R/R', 'Resistance//Resistance' )
ureg.unit( RatioScale(quantity.Resistance_ratio,'ohm_per_ohm','Ohm/Ohm') )

divider = qratio( r2,(r1+r2) )

v_in = qvalue( 5.12, volt)
v_out = qresult(divider * v_in)

if divider.unit.is_ratio_of(ohm.kind_of_quantity):
    print( "Resistive divider" )
    print( "  ratio =", divider )
    print( "  v_out =", v_out )
```

which produces the output

```
Resistive divider
  ratio = 0.832214765101 Ohm/Ohm
  v_out = 4.26093959732 V
```

Note, we use the operator `//` when declaring a dimensionless ratio as a kind of quantity. This is necessary to preserve information about the quantities in the ratio.

Another example is the voltage gain of an amplifying stage

```
from QV.prefix import micro

microvolt = ureg.unit( micro(volt) )

quantity.declare('Voltage_ratio','V/V','Voltage//Voltage')
volt_per_volt= ureg.unit( RatioScale(quantity.Voltage_ratio,'volt_per_volt','V/V') )

volt_per_millivolt = ureg.unit( proportional_unit(volt_per_volt,'volt_per_millivolt',
↪'V/mV',1E3) )
volt_per_microvolt = ureg.unit( proportional_unit(volt_per_volt,'volt_per_micovolt',
↪'V/uV',1E6) )

v1 = qvalue(0.5,volt)
v2 = qvalue(0.5,microvolt)
gain = qratio( v1, v2 )

print( "Gain =", qresult(gain) )
print( "Gain =", qresult(gain,volt_per_microvolt) )
print( "Gain =", qresult(gain,volt_per_millivolt) )
print( "Gain =", qresult(gain,volt_per_volt) )
```

The output is (Note, when no preferred unit is given (the first case), units are simplified to a dimensionless quantity.)

```
Gain = 1000000.0
Gain = 1.0 V/uV
Gain = 1000.0 V/mV
Gain = 1000000.0 V/V
```

3.5 Angles

It is well known that some SI quantities have the same dimensions and so cannot be distinguished by dimensional analysis [Brownstein]. In the case of angle, this ambiguity can be removed by introducing a new dimensional constant η but then some of the basic equations of physics also have to be changed [Quincey].

It is not as bad as it sounds. For example, the well-known equation

$$s = r \cdot \theta \; ,$$

for the length of arc subtended by an angle θ on a circle of radius r, becomes

$$s = \eta \cdot r \cdot \theta \; .$$

In this equation, θ has the dimension A and the constant η has the dimension A^{-1}, so s has the dimension of length, as expected (references [Brownstein] and [Quincey] should be consulted for more detail).

No one is suggesting that a dimension for angle should be added to the SI, however, a number of authors have remarked that using an extra dimension in computer systems would obtain more reliable dimensional homogeneity checks. The quantity-value package is perfect for this. The following simple example shows how the arc length calculation can be coded. More particularly, it shows how to introduce the dimension for angle and define the dimensional constant η.

```
quantity = Context( ("Length","L"), ("Time","T"), ("Angle","A") )
InverseAngle = quantity.declare('InverseAngle','1/A','1/A')

xi = UnitRegister("xi",quantity)

metre = xi.unit( RatioScale(quantity['Length'],'metre','m') )
second = xi.unit( RatioScale(quantity['Time'],'second','s')  )
radian = xi.unit( RatioScale(quantity['Angle'],'radian','rad')  )
inv_radian = xi.unit( RatioScale(quantity['InverseAngle'],'per radian','1/rad')  )

from math import pi

# Constants
PI = qvalue( pi, radian )
ETA = qresult( 1.0 / PI )

print( "pi =", PI)
print( "eta =", ETA )

radius = qvalue( 0.1, metre )
angle = qresult( PI/8 )
arc_length = qresult( ETA * angle * radius )

print( "arc length =", arc_length )
```

The output displays

```
pi = 3.14159265359 rad
eta = 0.318309886184 1/rad
arc length = 0.0125 m
```

CHAPTER

FOUR

MODULES

These are the package modules

4.1 Context

The *context* module provides support for calculations with quantities, in combination with the modules *signature* and *kind_of_quantity*. Kinds of quantity are associated with unique signatures in a context.

A *Context* is initialised by a set of quantities, which become a basis for that context. Other kinds of quantity can be declared by providing an expression that describes quantity in terms of the base quantities and possibly other quantities already declared.

For instance, in the following code block, resistance is declared in terms of voltage and current and power is declared in terms of voltage and resistance. The signature of power is $\mathrm{I}^1\mathrm{V}^1\mathrm{T}^0$, which is displayed by the print statement as `(1,1,0)`.

```
from QV import *

context = Context(
    ("Current","I"),("Voltage","V"),("Time","T")
)

context.declare('Resistance','R','Voltage/Current')
context.declare('Power','P','V*V/R')
print( context.signature('P') )
```

KindOfQuantity objects can be retrieved from a context and used in expressions:

```
Voltage = context['Voltage']
Resistance = context['Resistance']

tmp = Voltage/Resistance

print( tmp )
print( context.evaluate( tmp ) )
```

which displays

```
Div(V,R)
I
```

Here `tmp` is an intermediate result obtained by dividing objects representing voltage and current. QV does not automatically try to resolve intermediate results. The method *Context.evaluate()* must be used explicitly to resolve the kind of quantity of a temporary object.

class Context(**argv*)

A Context keeps a register of *KindOfQuantity* instances, and associates each with a unique signature.

A Context is initialised by a set of quantities. Other quantities can then be declared as products and quotients of these 'base' quantities, or of other derived quantities already declared.

The signature of declared quantities must be unique.

Example:

```
>>> context = Context( ("Length","L"),("Time","T") )
>>> context.declare('Speed','V','Length/Time')
KindOfQuantity('Speed','V')
```

property base_quantities

Return the base quantities in this context

declare(*koq_name*, *koq_symbol*, *expression*)

Declare a *KindOfQuantity* defined by `expression`

The `expression` may be products and quotients of *KindOfQuantity* objects, or a string representing these operations.

A `RuntimeError` is raised if the signature resulting from `expression` is already associated with a kind of quantity.

evaluate(*expression*)

Return the quantity represented by `expression`

The argument `expression` may be products and quotients of *KindOfQuantity* objects, or a string representing these operations.

A `RuntimeError` is raised if the signature of the result is not associated with a kind of quantity in the context.

signature(*koq*)

Return the signature associated with `koq`

4.2 Signature

The *Signature* class defines objects that hold a signature associated with a kind of quantity.

Every *Signature* is associated with a *Context*, in which a set of base quantities is defined.

Signature objects can be multiplied or divided.

Every *Signature* object has `numerator` and `denominator` members, which hold tuples of elements. Usually, the denominator contains zero values, and the object is said to be in *simplified* form. However, the denominator can be loaded with non-trivial element values by using the 'floor division' operator \\.

The 'floor division' operator \\ is an alternative to regular division. When floor division is used, the denominator of the *Signature* of the right-hand operand is added to the numerator of the left-hand operand and the numerator of the right-hand operand is added to the denominator of the left-hand operand. So, when the right and left-hand arguments have the same signatures, and both are in simplified form, a dimensionless ratio created by \\ retains information about the signatures of the original arguments. Regular division, on the other hand, subtracts the numerator of the right-hand operand from the numerator of the left-hand operand, and similarly for the denominator. So, when the right and left-hand arguments have the same signatures, the numerator and denominator of the resulting *Signature* will result in only zeros; the result is dimensionless with no information about the signatures of the original arguments.

class Signature(*context*, *numerator*, *denominator=()*)

A Signature has a pair of tuples that identify a kind of quantity.

Multiplication and division of signatures adds and subtracts the tuple elements, respectively.

The numerator and denominator of a Signature object are the tuples. This allows the signature of a 'dimensionless' quantity to be retained.

A Signature object is in 'simplified' form when the denominator is empty (or contains only zeros).

A Signature object may be converted to 'simplified' form by setting the numerator to the difference between the numerator and the denominator and setting the exponents in the denominator to zero.

A Signature refers to a `Context`, which contains a 1-to-1 mapping between signatures and kinds of quantity.

property context

The associated Context

property is_dimensionless

When elements in simplified form are all zero

is_ratio_of(*other*)

True when the object is a dimensionless ratio and the numerator has the same signature as the``other`` object.

property is_simplified

When elements in the denominator are all zero

simplify()

Return the signature in simplified form.

The numerator returned is the difference between the numerator and the denominator of this object, the elements in the denominator returned are all zero.

4.3 Kind of Quantity

A `KindOfQuantity` represents the general notion of a quantity, such as: a length, mass, speed, etc. This can be contrasted with more specific quantity definitions, like: my height, your weight, etc.

`KindOfQuantity` objects are defined (and identified) by a name and a term (a short name). For example, this code creates a new context with the base kinds of quantity Length and Time

```
from QV import *

context = Context( ("Length","L"), ("Time","T") )
Length, Time = context.base_quantities
```

The algebraic rules of quantity calculus are defined for `KindOfQuantity` objects (see, `context`).

class KindOfQuantity(*name*, *symbol*)

A type of quantity like mass, length, etc.

KindOfQuantity objects can be multiplied and divided. Declaring a ratio and simplifying a ratio is also supported.

4.4 Unit register

The unit register holds a collection of `RegisteredUnit` objects, which are a generalisation of the conventional notion of a measurement unit. Some `RegisteredUnit` objects are considered as reference units, the others are called related units. There can be only one reference unit for each kind of quantity, but any number of related units. Each related unit has a multiplier that can be used to convert a measure expressed in the related unit to a measure expressed in the reference unit.

The `unit_register` is associated with a `Context` to allow the validity of unit expressions to be checked by quantity calculus.

Units are declared by providing the name of the kind of quantity, the name of a scale (unit) and a term symbol (short name for the unit), for example

```
from QV import *

context = Context(('Length','L'))

SI =  UnitRegister("SI",context)
metre = SI.unit( RatioScale(context.Length,'metre','m') )   # reference unit
centimetre = SI.unit( prefix.centi(metre) ) # related unit
```

class UnitRegister(*name*, *context*)

A UnitRegister holds mappings between a kind-of-quantity, a type of scale and a collection of units.

A distinction is made between a reference unit and other related units for the same kind of quantity. There can be only one reference unit in the register for each kind of quantity.

conversion_from_A_to_B(*A*, *B*)

Return a conversion function for scale *A* to *B*

The function takes a single quantity-value argument *x* on *A* and returns a quantity-value result on *B*

conversion_function_values(*A*, *B*, **args*)

Register a function to convert from scale *A* to *B*

get(*koq*, *scale_type=<class 'QV.scale.RatioScale'>*)

reference_unit_for(*expr*)

Return the reference unit for *expr*

expr can be a product or quotient of registered-units or a product or quotient of kind-of-quantity objects or a registered-unit or a kind-of-quantity object.

unit(*scale*)

Register a new scale as a unit

The associated kind of quantity must not already have a scale with the same name or symbol

unit_dict_for(*expr*)

Return the units associated with the kind of quantity of *expr*

expr can be a product or quotient of registered-units, a product or quotient of kind-of-quantity objects, or a registered-unit or a kind-of-quantity object.

proportional_unit(*unit*, *name*, *symbol*, *conversion_factor*)

Declare a scale proportional to a unit already registered

4.5 Scale

The `scale` module contains the class `Scale` which implements generic scale behaviour. This class represents the conventional notion of a measurement scale (also commonly called a unit).

Different categories of scale are implemented as classes derived from `Scale`:

- Instances of `RatioScale` scales have an absolute zero, like the metre scale for length or the kelvin scale for thermodynamic temperature.
- `IntervalScale` scales are measurement scales with an arbitrary zero, like the Fahrenheit and Celsius temperature scales.

class IntervalScale(*kind_of_quantity*, *name*, *symbol*)

An `IntervalScale` has an arbitrary origin. Units associated with an interval scale may not be multiplied or divided.

static value_conversion_function()

Generic conversion function from one interval scale to another

class OrdinalScale(*kind_of_quantity*, *name*, *symbol*)

class RatioScale(*kind_of_quantity*, *name*, *symbol*, *conversion_factor=None*)

A `RatioScale` is a metric scale. Units may be multiplied and divided.

property conversion_factor

Return a conversion factor for a value on this scale to one on the reference scale

static value_conversion_function()

Generic conversion function from one ratio scale to another

class Scale(*kind_of_quantity*, *name*, *symbol*)

A Scale has a name (and a short name, or symbol) and contains a reference to the associated kind of quantity.

4.6 RegisteredUnit

The `registered_unit` module contains the class `RegisteredUnit` which implements generic unit behaviour.

The `UnitRegister` class handles the creation of `RegisteredUnit` instances.

class RegisteredUnit(*register*, *scale*)

A `RegisteredUnit` is associated with a `Scale` and a `UnitRegister`.

Multiplication and division of units is delegated to the scale and will be checked during execution.

The ‘floor’ division operator supports retention of information about the signature of ‘dimensionless’ quantities (ratios of the same kind of quantity).

conversion_to(*B*)

Return the conversion function from this unit to unit *B*

The conversion function takes a value argument *x* and returns the converted value on *B*

property is_dimensionless

True when the associated kind of quantity is dimensionless in the current context

is_ratio_of(*other_koq*)

True when the kind of quantity of `self` is a dimensionless ratio and the signature of the kind of quantity of `other_koq` match the numerator in the signature of the kind of quantity of `self`.

property is_simplified

True when the elements in the denominator of the associated kind of quantity are all zero

property kind_of_quantity
The kind of quantity

property register
The associated unit register

property scale
The scale

4.7 Quantity value

The term ‘quantity value’ refers to a measured value paired with the unit of measurement. `QV` provides several functions in the `quantity_value` module that support the notion of quantity values:

- `qvalue()` creates a quantity-value,
- `qratio()` creates a quantity-value that is a dimensionless ratio,
- `qresult()` resolves the unit for an expression involving quantity-values.

More information is given in the *Examples* section.

qratio(*value_unit_1*, *value_unit_2*, *unit=None*)

Return a quantity value for `value_unit_1/value_unit_2`. If the signature of the associated units are in simplified form, signature information is retained in the quotient.

When `unit` is None, the reference unit is used.

Example

```
>>> context = Context( ("Current","I"),("Voltage","V") )
>>> ureg = UnitRegister("ureg",context)
>>> volt = ureg.unit( RatioScale(context['Voltage'],'volt','V') )
>>> voltage_ratio = context.declare('voltage_ratio','V/V','Voltage//Voltage')
>>> volt_per_volt = ureg.unit( RatioScale(context['voltage_ratio'],'volt_per_volt
↪','V/V') )
>>> v1 = qvalue(1.23, volt)
>>> v2 = qvalue(9.51, volt)
>>> qratio( v2,v1 )
qvalue(7.73170731...,volt_per_volt)
```

qresult(*value_unit*, *unit=None*, *simplify=True*, *value_result=<function <lambda>>*, *arg, **kwarg)

Return a `qvalue`.

`value_unit` is a quantity-value or expression of quantity-values.

If a `unit` is supplied, it is used to report the measure. If not, the measure is reported in the reference unit for that quantity.

If `simplify` is `True`, unit signatures will be simplified.

The function `value_result` is applied to the value as a final processing step.

Example

```
>>> context = Context( ("Length","L"), ("Time","T") )
>>> Speed = context.declare('Speed','V','Length/Time')
>>> si =  UnitRegister("si",context)
>>> metre = si.unit( RatioScale(context['Length'],'metre','m') )
>>> second = si.unit( RatioScale(context['Time'],'second','s') )
>>> metre_per_second = si.unit( RatioScale(context['Speed'],'metre_per_second',
↪'m*s-1') )
```

(continues on next page)

(continued from previous page)

```
>>> d = qvalue(0.5,metre)
>>> t = qvalue(1.0,second)
>>> v0 = qresult(d/t)
>>> print( "average speed =", v0 )
average speed = 0.5 m*s-1
>>> x0 = qvalue(.3,metre)
>>> print( "displacement =", x0 + v0*t )
displacement = 0.8 m
```

qvalue(*value*, *unit*)

Create a new quantity value object.

`value` is the measure, `unit` is the measurement scale

Example

```
>>> context = Context( ("Length","L"), ("Time","T") )
>>> si = UnitRegister("si",context)
>>> metre = si.unit( RatioScale(context['Length'],'metre','m') )
>>> qvalue( 1.84, metre )
qvalue(1.84,metre)
```

unit(*quantity_value*)

Return the unit (measurement scale)

value(*quantity_value*)

Return the value

4.8 Prefix

The *prefix* module defines standard metric and binary prefixes: `yocto` through to `yotta` and `kibi` through to `yobi`. There is also a function to generate a set of SI mass units (as this is a special case, with the kilogram being the conventional reference unit).

si_mass_units(*kg_unit*)

Generate multiples and sub-multiples for SI mass units

`kg_unit` must be defined, with name `kilogram` and symbol `kg`

Example:

```
>>> context = Context( ('Mass','M') )
>>> SI =  UnitRegister("SI",context)
>>> kilogram = SI.unit( RatioScale(context['Mass'],'kilogram','kg') )
>>> prefix.si_mass_units(kilogram)
>>> print( SI.Mass.gram.scale.name )
gram
>>> print( repr(SI.Mass.gram) )
RegisteredUnit(KindOfQuantity('Mass','M'),'gram','g')
```

metric_prefixes

A collection of all metric prefixes.

Useful for generating all related units by iteration:

```
>>> context = Context( ('Time','T') )
>>> second = SI.unit( RatioScale(context['Time'],'second','s') )
```

(continues on next page)

(continued from previous page)

```
>>> for p_i in prefix.metric_prefixes:
...     prefixed_scale = p_i(second.scale)
...     print( "{0.name} ({0.symbol}): {0.prefix:.1E}".format(prefixed_scale) )
...
yoctosecond (ys): 1.00E-24
zeptosecond (zs): 1.00E-21
attosecond (as): 1.00E-18
femtosecond (fs): 1.00E-15
picosecond (ps): 1.00E-12
nanosecond (ns): 1.00E-09
microsecond (us): 1.00E-06
millisecond (ms): 1.00E-03
centisecond (cs): 1.00E-02
decisecond (ds): 1.00E-01
dekasecond (das): 1.00E+01
hectosecond (hs): 1.00E+02
kilosecond (ks): 1.00E+03
megasecond (Ms): 1.00E+06
gigasecond (Gs): 1.00E+09
terasecond (Ts): 1.00E+12
petasecond (Ps): 1.00E+15
exasecond (Es): 1.00E+18
zettasecond (Zs): 1.00E+21
yottasecond (Ys): 1.00E+24
```

binary_prefixes

A collection of binary prefixes.

Useful for generating all related units by iteration:

```
>>> byte_scale = RatioScale('Data','byte','b')
>>> for p_i in prefix.binary_prefixes:
...     prefixed_scale = p_i(byte_scale)
...     print( "{0.name} ({0.symbol}): {0.prefix}".format(prefixed_scale) )
...
kibibyte (kib): 1048
mebibyte (Mib): 1098304
gibibyte (Gib): 1151022592
tebibyte (Tib): 1206271676416
pebibyte (Pib): 1264172716883968
exbibyte (Eib): 1324853007294398464
zebibyte (Zib): 1388445951644529590272
yottabyte (Yib): 1455091357323467010605056
```

4.9 UnitsDict

The `units_dict` module defines `UnitsDict`, which implements a mapping for unit names, and unit symbols, to unit objects.

class UnitsDict(**args*, ***kwargs*)

A dictionary-like mapping of names, and short names (symbols), to objects representing units.

The names and short names are keys. They are unique and cannot be overwritten once defined (but, they can be deleted).

CHAPTER

FIVE

RELEASE NOTES

5.1 Release Notes

5.1.1 Version 0.2.0 (2021-04-30)

- Terminology has changed substantially. Use of the term 'dimension' in version 0.1.0 has been reconsidered and in many cases we now write 'signature' instead. This allows us to use the term 'dimension' correctly.
- The `Dimension` class has been renamed *`Signature`*. The structure of a Signature class is now described in terms of a ratio of tuples containing numerical elements (rather than a ratio of dimensions composed of exponents).
- The `Unit` class has been renamed *`RegisteredUnit`* and placed in its own module, *`registered_unit`*.
- A hierarchy of scales classes is now provided: *`Scale`*, *`OrdinalScale`*, *`IntervalScale`* and *`RatioScale`* The *`RatioScale`* represents the behaviour of the units that were represented in the previous release (metric units).
- Both *`IntervalScale`* and *`RatioScale`* have `conversion_function()` methods that return a generic function for conversion between different scales for the same quantities. A new method *`UnitRegister.conversion_function_values()`* can be used to register the specific parameters required for value conversion between two scales using the *conversion_function*.
- Support for Python 2 has been dropped.

5.1.2 Version 0.1.0 (2020-02-24)

- First release.

5.2 Indices and tables

- genindex
- modindex

BIBLIOGRAPHY

[VIM_S1.2] The international vocabulary of metrology—basic and general concepts and associated terms (section 1.2), online: https://www.bipm.org/en/publications/guides/.

[VIM_S1.19] The international vocabulary of metrology—basic and general concepts and associated terms (section 1.19), online: https://www.bipm.org/en/publications/guides/.

[SI_brochure] The International System of Units (SI): https://www.bipm.org/en/publications/si-brochure/

[Brownstein] K. R. Brownstein, *Angles - lets treat them squarely*, Am. J. Phys. 65(7), July 1997, pp 605-614 .

[Quincey] P. Quincey and R. J. C. Brown, *Implications of adopting plane angle as a base quantity in the SI*, Metrologia 53, 2016, pp 998-1002.

PYTHON MODULE INDEX

INDEX