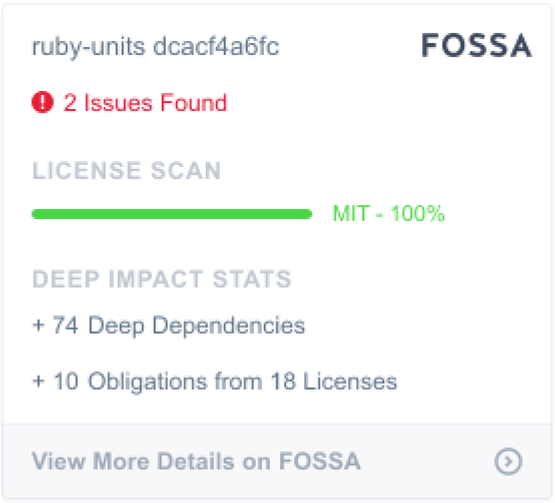
Ruby Units


ruby-units dcacf4a6fc
FOSSA
2 Issues Found
LICENSE SCAN
MIT - 100%
DEEP IMPACT STATS
+ 74 Deep Dependencies
+ 10 Obligations from 18 Licenses
View More Details on FOSSA

maintainability D test coverage 97%

Kevin C. Olbrich, Ph.D.

Project page: http://github.com/olbrich/ruby-units

Introduction

Many technical applications make use of specialized calculations at some point. Frequently, these calculations require unit conversions to ensure accurate results. Needless to say, this is a pain to properly keep track of, and is prone to numerous errors.

Solution

The 'Ruby units' gem is designed to simplify the handling of units for scientific calculations. The units of each quantity are specified when a Unit object is created and the Unit class will handle all subsequent conversions and manipulations to ensure an accurate result.

Installation

This package may be installed using:

```
gem install ruby-units
```

or add this to your `Gemfile`

```
gem 'ruby-units'
```

Usage

```
unit = Unit.new("1")                 # constant only
unit = Unit.new("mm")                # unit only (defaults to a scalar
    of 1)
unit = Unit.new("1 mm")              # create a simple unit
unit = Unit.new("1 mm/s")            # a compound unit
unit = Unit.new("1 mm s^-1")         # in exponent notation
unit = Unit.new("1 kg*m^2/s^2")      # complex unit
unit = Unit.new("1 kg m^2 s^-2")     # complex unit
unit = Unit.new("1 mm")              # shorthand
unit = "1 mm".to_unit                # convert string object
unit = object.to_unit                # convert any object using object.
    to_s
unit = Unit.new('1/4 cup')           # Rational number
unit = Unit.new('1+1i mm')           # Complex Number
```

Rules

1. only 1 quantity per unit (with 2 exceptions… 6’5” and ‘8 lbs 8 oz’)
2. use SI notation when possible
3. spaces in units are allowed, but ones like ‘11/m’ will be recognized as ‘11 1/m’.

Unit compatibility

Many methods require that the units of two operands are compatible. Compatible units are those that can be easily converted into each other, such as ‘meters’ and ‘feet’.

```
unit1 =~ unit2                  #=> true if units are compatible
unit1.compatible?(unit2)        #=> true if units are compatible
```

Unit Math

```
Unit#+()      # Add. only works if units are compatible
Unit#-()      # Subtract. only works if units are compatible
Unit#*()      # Multiply.
Unit#/()      # Divide.
Unit#**()     # Exponentiate.  Exponent must be an integer, can be
    positive,  negative, or zero
Unit#inverse  # Returns 1/unit
Unit#abs      # Returns absolute value of the unit quantity.  Strips
    off the  units
Unit#ceil     # rounds quantity to next highest integer
```

```
Unit#floor    # rounds quantity down to next lower integer
Unit#round    # rounds quantity to nearest integer
Unit#to_int   # returns the quantity as an integer
```

Unit will coerce other objects into a Unit if used in a formula. This means…

```
Unit.new("1 mm") + "2 mm"  == Unit.new("3 mm")
```

This will work as expected so long as you start the formula with a `Unit` object.

Conversions & Comparisons

Units can be converted to other units in a couple of ways.

```
unit.convert_to('ft')             # convert
unit1 = unit >> "ft"              # convert to 'feet'
unit >>= "ft"                     # convert and overwrite original
    object
unit3 = unit1 + unit2             # resulting object will have the
    units of unit1
unit3 = unit1 - unit2             # resulting object will have the
    units of unit1
unit1 <=> unit2                   # does comparison on quantities in
    base units, throws an exception if not compatible
unit1 === unit2                   # true if units and quantity are the
    same, even if 'equivalent' by <=>
unit1 + unit2 >> "ft"             # converts result of math to 'ft'
(unit1 + unit2).convert_to('ft')  # converts result to 'ft'
```

Any object that defines a `to_unit` method will be automatically coerced to a unit during calculations.

Text Output

Units will display themselves nicely based on the display_name for the units and prefixes. Since `Unit` implements a `Unit#to_s`, all that is needed in most cases is:

```
"#{Unit.new('1 mm')}"  #=> "1 mm"
```

The `to_s` also accepts some options.

```
Unit.new('1.5 mm').to_s("%0.2f")  # "1.50 mm".  Enter any valid format
                                  #   string.  Also accepts strftime
                                     format
Unit.new('10 mm').to_s("%0.2f in")# "0.39 in". can also format and
    convert in
```

```

Unit.new('1.5 mm').to_s("in")     # the same time.
                                  # converts to inches before printing
Unit.new("2 m").to_s(:ft)         # returns 6'7"
Unit.new("100 kg").to_s(:lbs)     # returns 220 lbs, 7 oz
Unit.new("100 kg").to_s(:stone)   # returns 15 stone, 10 lb
```

Time Helpers

`Time`, `Date`, and `DateTime` objects can have time units added or subtracted.

```
Time.now + Unit.new("10 min")
```

Several helpers have also been defined. Note: If you include the 'Chronic' gem, you can specify times in natural language.

```
Unit.new('min').since(DateTime.parse('9/18/06 3:00pm'))
```

Durations may be entered as 'HH:MM:SS, usec' and will be returned in 'hours'.

```
Unit.new('1:00')      #=> 1 h
Unit.new('0:30')      #=> 0.5 h
Unit.new('0:30:30')   #=> 0.5 h + 30 sec
```

If only one ":" is present, it is interpreted as the separator between hours and minutes.

Ranges

```
[Unit.new('0 h')..Unit.new('10 h')].each {|x| p x}
```

works so long as the starting point has an integer scalar

Math functions

All Trig math functions (sin, cos, sinh, hypot...) can take a unit as their parameter. It will be converted to radians and then used if possible.

Temperatures

Ruby-units makes a distinction between a temperature (which technically is a property) and degrees of temperature (which temperatures are measured in).

Temperature units (i.e., ‘tempK’) can be converted back and forth, and will take into account the differences in the zero points of the various scales. Differential temperature (e.g., Unit.new(‘100 degC’)) units behave like most other units.

```
Unit.new('37 tempC').convert_to('tempF')    #=> 98.6 tempF
```

Ruby-units will raise an exception if you attempt to create a temperature unit that would fall below absolute zero.

Unit math on temperatures is fairly limited.

```
Unit.new('100 tempC') + Unit.new('10 degC')    # '110 tempC'.to_unit
Unit.new('100 tempC') - Unit.new('10 degC')    # '90 tempC'.to_unit
Unit.new('100 tempC') + Unit.new('50 tempC')   # exception (can't add
    two temperatures)
Unit.new('100 tempC') - Unit.new('50 tempC')   # '50 degC'.to_unit (get
    the difference between two temperatures)
Unit.new('50 tempC')  - Unit.new('100 tempC') # '-50 degC'.to_unit
Unit.new('100 tempC') * scalar                 # '100*scalar tempC'.
    to_unit
Unit.new('100 tempC') / scalar                 # '100/scalar tempC'.
    to_unit
Unit.new('100 tempC') * unit                   # exception
Unit.new('100 tempC') / unit                   # exception
Unit.new('100 tempC') ** N                     # exception

Unit.new('100 tempC').convert_to('degC')   #=> Unit.new('100 degC')
```

This conversion references the 0 point on the scale of the temperature unit

```
Unit.new('100 degC').convert_to('tempC')   #=> '-173 tempC'.to_unit
```

These conversions are always interpreted as being relative to absolute zero. Conversions are probably better done like this…

```
Unit.new('0 tempC') + Unit.new('100 degC') #=> Unit.new('100 tempC')
```

Defining Units

It is possible to define new units or redefine existing ones.

Define New Unit The easiest approach is to define a unit in terms of other units.

```
Unit.define("foobar") do |foobar|
  foobar.definition   = Unit.new("1 foo") * Unit.new("1 bar")   #
      anything that results in a Unit object
```

```ruby
  foobar.aliases      = %w{foobar fb}                # array of
      synonyms for the unit
  foobar.display_name = "Foobar"                     # How unit is
      displayed when output
end
```

Redefine Existing Unit Redefining a unit allows the user to change a single aspect of a definition without having to re-create the entire definition. This is useful for changing display names, adding aliases, etc.

```ruby
Unit.redefine!("cup") do |cup|
  cup.display_name  = "cup"
end
```

Useful methods

1. `scalar` will return the numeric portion of the unit without the attached units
2. `base_scalar` will return the scalar in base units (SI)
3. `units` will return the name of the units (without the scalar)
4. `base` will return the unit converted to base units (SI)

Storing in a database

Units can be stored in a database as either the string representation or in two separate columns defining the scalar and the units. Note that if sorting by units is desired you will want to ensure that you are storing the scalars in a consistent unit (i.e, the base units).

Namespaced Class

Sometimes the default class 'Unit' may conflict with other gems or applications. Internally ruby-units defines itself using the RubyUnits namespace. The actual class of a unit is the RubyUnits::Unit. For simplicity and backwards compatibility, the `::Unit` class is defined as an alias to `::RubyUnits::Unit`.

To load ruby-units without this alias…

```ruby
require 'ruby_units/namespaced'
```

When using bundler…

```ruby
gem 'ruby-units', require: 'ruby_units/namespaced'
```

Note: when using the namespaced version, the `Unit.new('unit string')` helper will not be defined.

Configuration

Configuration options can be set like:

```ruby
RubyUnits.configure do |config|
  config.separator = false
end
```

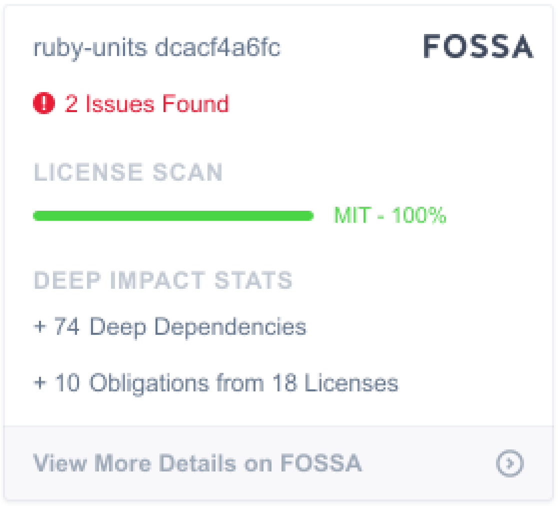
Currently there is only one configuration you can set:

1. separator (true/false): should a space be used to separate the scalar from the unit part during output.

NOTES

Performance vs. Accuracy Ruby units was originally intended to provide a robust and accurate way to do arbitrary unit conversions. In some cases, these conversions can result in the creation and garbage collection of a lot of intermediate objects during calculations. This in turn can have a negative impact on performance. The design of ruby-units has emphasized accuracy over speed. YMMV if you are doing a lot of math involving units.

License


ruby-units dcacf4a6fc

FOSSA

2 Issues Found

LICENSE SCAN

MIT - 100%

DEEP IMPACT STATS

+ 74 Deep Dependencies

+ 10 Obligations from 18 Licenses

View More Details on FOSSA