Unum

Lukasz Laba

Jan 22, 2022

CONTENTS

If you make scientific or engineering calculations it is a best practice to always include units when you define variables if they are relevant. For example, if you define velocity to be 100, and time to be 0.5, Python cannot perform unit checking or unit balancing when you calculate the distance. It is a better practice to define the velocity as 100 km/h and the time as 30 min. It can also prevent errors later on. There are few ‘Famous Unit Conversion Errors’ that occurred because of unit calculation issue.

Since the pure Python has no feature that let you make unit calculation you neeed additional package for that. Unum can handle units for you, there’s a large list of built-in units available for the most common applications that could be easily extended by the user. Unum automatically carries units forward throughout calculations, always checking to make sure that the units being used in formulas and equations are dimensionally compatible. It warns you if you try to perform math with quantities in incompatible units. For example, you get an error notification if you try to add quantities of length, temperature, time, and energy together.

CHAPTER

ONE

USER GUIDE

1.1 Introduction

Unum stands for ‘unit-numbers’. It is a Python module that allows you to define and manipulate quantities with units attached such as 60 seconds, 500 watts, 42 miles-per-hour, 100 kg per square meter, 14400 bits per second, 30 dollars, and so on.

Features include:

- Exceptions for incorrect use of units.
- Automatic and manual conversion between compatible units.
- Easily extended to arbitrary units.
- Integration with any type supporting arithmetic operations, including Numpy arrays and standard library types like complex and fractions.Fraction.
- Customizable output formatting.

1.2 Example

For a simple example, let’s can calculate Usain Bolt’s average speed during his record-breaking performance in the 2008 Summer Olympics:

```
>>> from unum.units import * # Load a number of common units.
>>> distance = 100*m
>>> time = 9.683*s
>>> speed = distance / time
>>> speed
10.3273778788 [m/s]
>>> speed.asUnit(mile/h)
23.1017437978 [mile/h]
```

If we do something dimensionally incorrect, we get an exception rather than silently computing a correct result. Let’s try calculating his kinetic energy using an erroneous formula:

```
>>> KE = 86*kg * speed / 2 # Should be speed squared!
>>> KE.asUnit(J)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unum\__init__.py", line 171, in asUnit
```

(continues on next page)

(continued from previous page)

```
    s, o = self.matchUnits(other)
  File "unum\__init__.py", line 258, in matchUnits
    raise IncompatibleUnitsError(self, other)
unum.IncompatibleUnitsError: [kg.m/s] can't be used with [J]
```

The exception pinpoints the problem, allowing us to examine the units and fix the formula:

```
>>> KE = 86*kg * speed**2 / 2
>>> KE.asUnit(J)
4586.15355558 [J]
```

Unum will also report errors in attempting to add incompatible units:

```
>>> 1*s + 2*kg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unum\__init__.py", line 269, in __add__
    s, o = self.matchUnits(Unum.coerceToUnum(other))
  File "unum\__init__.py", line 258, in matchUnits
    raise IncompatibleUnitsError(self, other)
unum.IncompatibleUnitsError: [s] can't be converted to [kg]
```

and when units are present in operations that don't expect them, such as the second part of an exponentiation:

```
>>> 2 ** (2*m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unum\__init__.py", line 398, in __rpow__
    return Unum.coerceToUnum(other).__pow__(self)
  File "unum\__init__.py", line 332, in __pow__
    other.checkNoUnit()
  File "unum\__init__.py", line 230, in checkNoUnit
    raise ShouldBeUnitlessError(self)
unum.ShouldBeUnitlessError: expected unitless, got 2 [m]
```

Unums are automatically coerced back to regular numbers where legal and desirable. Here log10 expects a plain integer or float:

```
>>> math.log10((1000 * m) / (10 * m)) # Units cancel, so it's ok.
2.0
```

1.3 Installation

Unum is pure python package. Python 2.2 or higher needed. Python 3.x work as well.

Unum is available through PyPI and can be easy installed using `pip` command

```
pip install unum
```

Alternately, you can obtain the source code and type:

```
python setup.py install
```

in the source code directory.

1.4 Usage

Unums are ordinary Python objects and support all the mathematical operations available in Python using the same syntax as usual:

```
>>> 1*m + 2*m
3 [m]
>>> 3*m * 4*m
12 [m2]
>>> abs(-5*m)
5 [m]
>>> 6*m > 5*m
True
>>> 5*m ** 3
5 [m3]
>>> (5*m) ** 3
125 [m3]
```

Note how the parentheses in the last example makes the exponentiation apply to the whole number rather than just the "m".

If you are using Python 2.x, be very careful with the way division works:

```
>>> 1 / 3 * (m/s)
0 [m/s]
>>> 1.0 / 3 * (m/s)
0.333333333333 [m/s]
```

Dividing two integers truncates the remainder to produce another integer, while dividing two floats produces another float. In Python 3.x, division with the / operator always produces a float, and the // operator always performs integer division.

It's possible to have Unums where all the units have cancelled; these are conceptually the same as a raw number, and can be used accordingly:

```
>>> two = (2 * m) / m
>>> two
2 []
>>> 5**two
25 []
>>> import math
>>> math.log(two)
0.69314718055994529
```

What's happening here is that when math.log wants a plain number, it coerces (converts) the Unum into a plain number. You can do this manually using Python's builtin functions:

```
>>> int(two)
2
```

(continues on next page)

(continued from previous page)

```
>>> float(two)
2.0
```

Another way to get at the value inside the Unum is with the asNumber method, which allows you to do a conversion at the same time:

```
>>> speed.asNumber(mile/h) # Get the value in mile/h
23.101743797879877
>>> speed.asNumber() # Get the value in the current units
10.3273778788
```

1.5 Standard library integration

The standard library types complex and Fraction can be used with Unum transparently:

```
>>> length = 1j * m # One imaginary meter.
>>> length
1j [m]
>>> length ** 2 # j * j == -1
(-1+0j) [m2]

>>> from fractions import Fraction
>>> Fraction(1, 3) * S
1/3 [s]
>>> Fraction(1,2) * S + Fraction(1,3) * S
5/6 [s]
```

Unums are picklable, so you can store them into files or databases as usual; see the "pickle" and "shelve" modules in the Python standard library for more details.

1.6 Numpy integration

Unum works with Numpy with a couple caveats. First, there is a difference between left-multiplying and right-multiplying with an Unum:

```
>>> from numpy import array
>>> array([2,3,4]) * m  # note that meters is on the right here
array([2 [m], 3 [m], 4 [m]], dtype=object)
>>> m * array([2,3,4])  # this time meters is on the left
[2 3 4] [m]
```

Right-multiplying produces an array of Unum objects, which is often undesirable since each Unum object takes up more memory than a simple number does. However, this does allow the objects to be different types, if you so desire.

Generally, a better idea is to use left-multiplication, which produces a single Unum object containing the array as its value. This is memory-efficient, but constrains all the objects in the array to be the same type.

Another way to get the effect of left-multiplication is to use the provided unum.uarray helper function, which turns an array-like object into a unitless Unum, which you can then multiply on the right as normal:

```
>>> from unum import uarray
>>> uarray([2,3,4])
[2 3 4] []
>>> uarray([2,3,4]) * m
[2 3 4] [m]
```

The second caveat is most of NumPy's universal functions don't work on Unums, even if they are unitless. Arithmetic operators work, but trigonometric functions do not:

```
>>> lengths = m * [2,3,4]
>>> lengths
[2, 3, 4] [m]
>>> length + 1
[3, 4, 5] [m]
>>> cos(lengths)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: cos
```

Luckily, you can extract the value of any Unum using the asNumber method, allowing you to use the array inside:

```
>>> cos(lengths.asNumber())
array([-0.41614684, -0.9899925 , -0.65364362])
```

If anyone has ideas on improving integration with Unum, I'd love to hear from you.

1.7 Defining New Units

Creating new units is done with a single function call. Imagine you want to define a new unit called 'spam', with derived units 'kilospam', 'millispam', and 'sps' (spam per second):

```
>>> from unum import Unum
>>> SPAM = Unum.unit('spam')
```

Now the variable SPAM refers to a Unum representing one 'spam'. The name of the variable is arbitrary, and the same Unum can have multiple names:

```
>>> spam = SPAM
>>> spam
1 [spam]
```

Here both spam and SPAM can be used interchangeably to refer to the same thing. Derived units are defined in relation to this base unit:

```
>>> KSPAM = Unum.unit('kilospam', 1000 * SPAM)
>>> MSPAM = Unum.unit('millispam', 0.001 * SPAM)
>>> SPS = Unum.unit('sps', SPAM / S)
```

The second argument provided is the definition of the derived unit in terms of previously defined units. Note that the variable name is arbitrary and independent of the longer symbol used. Now you can work with 'spammed' quantities.

```
>>> (500 * MSPAM).asUnit(SPAM)
0.5 [spam]
>>> (5000 * MSPAM).asUnit(SPAM)
5.0 [spam]
>>> SPS.asUnit(MSPAM/S)
1000.0 [millispam/s]
>>> 5*SPS * 20*S
100 [spam]
>>> (10*SPS)**2
100 [sps2]
```

1.8 Importing units

You can keep your favorite units in a normal Python module, and then import that module to have them available anywhere. A module containing your 'spam' units could be as simple as:

```
# my_spam.py
from unum.units import *
from unum import Unum

SPAM = Unum.unit('spam')
KSPAM = Unum.unit('kilospam', 1000 * SPAM)
MSPAM = Unum.unit('millispam', 0.001 * SPAM)
SPS = Unum.unit('sps', SPAM / S)
```

Placing this module anywhere on your Python path will allow you to do:

```
>>> from my_spam import *
```

and have your units available.

1.9 Beware of the name conflict issue

It is a good practice to use:

```
import unum.units as u
```

instead of:

```
from unum.units import *
```

In that case all your units will be holded inside `u` object so you can still define for example `m` and `g` variables during your calculations and not afraid that you overwrite the meter and gram unit definition.

```
>>> import unum.units as u
>>> m = 2.3*u.kg
>>> g = 10 * u.m/u.s**2
>>> F = m * g
>>> F
23.0 [kg.m/s2]
```

(continues on next page)

(continued from previous page)

```
>>> F.asUnit(u.N)
>>> F.asUnit(u.N)
23.0 [N]
```

1.10 Predefined units

Unum comes with the standard SI units as well as some other widely used units. You can browse the “units” folder in the “unum” folder to see what’s available. If you want to contribute more units, feel free to submit them.

1.11 Advanced usage

1.11.1 Custom formatting

The string representation of Unums can be configured by modifying the variables of the Unum class:

```
>>> Unum.UNIT_SEP = ' '
>>> Unum.UNIT_DIV_SEP = None
>>> Unum.UNIT_FORMAT = '%s'
>>> Unum.UNIT_HIDE_EMPTY = True
>>> Unum.VALUE_FORMAT = "%15.7f"
>>> M
>>>      1.0000000 m
>>> 25 * KG*M/S**2
     25.0000000 kg m s-2
>>> M/ANGSTROM
10000000000.0000000
>>>
```

See the docstrings in the class for more detail.

1.11.2 Normalization

By default, Unum will find the shortest unit representation among equivalent expressions, by applying the known unit conversion rules. This is called normalization. For example a pressure given in Pascal multiplied by a surface will give a force in Newton, since one Pascal is equal, by definition, to a Newton per square meter:

```
>>> Pa * m**2
1 [N]
```

This behavior can be controlled by a flag on the Unum class:

```
>>> Unum.AUTO_NORM = False
>>> Pa * m**2
1 [Pa.m2]
```

Then you must manually normalize by calling the normalize method:

```
>>> x = Pa * m**2
>>> x
1 [Pa.m2]
>>> x.normalize()
1 [N]
>>> x
1 [N]
```

Note that normalize permanently modifies the instance itself as a side-effect.

1.12 Porting from older Unum versions

See the README for changes to the API from Unum 4.0. While most things should still work, there are a couple important changes to be aware of.

CHAPTER

TWO

ABOUT

2.1 License

2.2 Source code

Code repository: https://bitbucket.org/lukaszlaba/unum

2.3 Contact

Contact: Lukasz Laba <lukaszlaba@gmail.com>