python-measurement Documentation

Release 1.0

Adam Coddington

Jan 09, 2018

Contents

Easily use and manipulate unit-aware measurement objects in Python.

django.contrib.gis.measure has these wonderful 'Distance' objects that can be used not only for storing a unit-aware distance measurement, but also for converting between different units and adding/subtracting these objects from one another.

This module not only provides those Distance and Area measurement objects (courtesy of Django), but also other measurements including Weight, Volume, and Temperature.

Warning: Measurements are stored internally by converting them to a floating-point number of a (generally) reasonable SI unit. Given that floating-point numbers are very slightly lossy, you should be aware of any inaccuracies that this might cause.

TLDR: Do not use this in navigation algorithms guiding probes into the atmosphere of extraterrestrial worlds.

Contents:

CHAPTER 1

Creating your own Measure Class

You can create your own measures easily by subclassing either `measurement.base.MeasureBase` or `measurement.base.BidimensionalMeasure`.

1.1 Simple Measures

If your measure is not a measure dependent upon another measure (e.g speed, distance/time) you can create new measurement by creating a subclass of `measurement.base.MeasureBase`.

A simple example is Weight:

```
from measurement.base import MeasureBase

class Weight(MeasureBase):
    STANDARD_UNIT = 'g'
    UNITS = {
        'g': 1.0,
        'tonne': 1000000.0,
        'oz': 28.3495,
        'lb': 453.592,
        'stone': 6350.29,
        'short_ton': 907185.0,
        'long_ton': 1016000.0,
    }
    ALIAS = {
        'gram': 'g',
        'ton': 'short_ton',
        'metric tonne': 'tonne',
        'metric ton': 'tonne',
        'ounce': 'oz',
        'pound': 'lb',
        'short ton': 'short_ton',
        'long ton': 'long_ton',
```

```
    }
    SI_UNITS = ['g']
```

Important details:

- `STANDARD_UNIT` defines what unit will be used internally by the library for storing the value of this measurement.
- `UNITS` provides a mapping relating a unit of your `STANDRD_UNIT` to any number of defined units. In the example above, you will see that we've established `28.3495 g` to be equal to `1 oz`.
- `ALIAS` provides a list of aliases mapping keyword arguments to `UNITS`. these values are allowed to be used as keyword arguments when either creating a new unit or guessing a measurement using `measurement.utils.guess`.
- `SI_UNITS` provides a list of units that are SI Units. Units in this list will automatically have new units and aliases created for each of the main SI magnitudes. In the above example, this causes the list of `UNITS` and `ALIAS` es to be extended to include the following units (aliases): `yg` (yottagrams), `zg` (zeptograms), `ag` (attograms), `fg` (femtograms), `pg` (picograms), `ng` (nanograms), `ug` (micrograms), `mg` (milligrams), `kg` (kilograms), `Mg` (megagrams), `Gg` (gigagrams), `Tg` (teragrams), `Pg` (petagrams), `Eg` (exagrams), `Zg` (zetagrams), `Yg` (yottagrams).

1.1.1 Using formula-based conversions

In some situations, your conversions between units may not be simple enough to be accomplished by using simple conversions (e.g. temperature); for situations like that, you should use `sympy` to create expressions relating your measure's standard unit and the unit you're defining:

```
from sympy import S, Symbol
from measurement.base import MeasureBase

class Temperature(MeasureBase):
    SU = Symbol('kelvin')
    STANDARD_UNIT = 'k'
    UNITS = {
        'c': SU - S(273.15),
        'f': (SU - S(273.15)) * S('9/5') + 32,
        'k': 1.0
    }
    ALIAS = {
        'celsius': 'c',
        'fahrenheit': 'f',
        'kelvin': 'k',
    }
```

Important details:

- See above 'Important Details' under *Normal Measures*.
- `SU` must define the symbol used in expressions relating your measure's `STANDARD_UNIT` to the unit you're defining.

1.2 Bi-dimensional Measures

Some measures are really just compositions of two separate measures – Speed, being a measure of the amount of distance covered over a unit of time, is one common example of such a measure.

You can create such measures by subclassing `measurement.base.BidimensionalMeasure`.

```
from measurement.base import BidimensionalMeasure

from measurement.measures.distance import Distance
from measurement.measures.time import Time


class Speed(BidimensionalMeasure):
    PRIMARY_DIMENSION = Distance
    REFERENCE_DIMENSION = Time

    ALIAS = {
        'mph': 'mi__hr',
        'kph': 'km__hr',
    }
```

Important details:

- `PRIMARY_DIMENSION` is a class that measures the variable dimension of this measure. In the case of 'miles-per-hour', this would be the 'miles' or 'distance' dimension of the measurement.
- `REFERENCE_DIMENSION` is a class that measures the unit (reference) dimension of the measure. In the case of 'miles-per-hour', this would be the 'hour' or 'time' dimension of the measurement.
- `ALIAS` defines a list of convenient abbreviations for use either when creating or defining a new instance of this measurement. In the above case, you can create an instance of speed like `Speed(mph=10)` (equivalent to `Speed(mile__hour=10)`) or convert to an existing measurement (`speed_measurement`) into one of the aliased measures by accessing the attribute named – `speed_measurement.kph` (equivalent to `speed_measurement.kilometer__hour`).

Note: Although unit aliases defined in a bi-dimensional measurement's `ALIAS` dictionary can be used either as keyword arguments or as attributes used for conversion, unit aliases defined in simple measurements (those subclassing `measurement.base.MeasureBase`) can be used only as keyword arguments.

CHAPTER 2

Installation

You can either install from pip:

```
pip install measurement
```

or checkout and install the source from the github repository:

```
git clone https://github.com/coddingtonbear/python-measurement.git
cd python-measurement
python setup.py install
```

CHAPTER 3

Measures

This application provides the following measures:

Note: Python has restrictions on what can be used as a method attribute; if you are not very familiar with python, the below chart outlines which units can be used only when creating a new measurement object ('Acceptable as Arguments') and which are acceptable for use either when creating a new measurement object, or for converting a measurement object to a different unit ('Acceptable as Arguments or Attributes')

Units that are acceptable as arguments (like the distance measurement term `km`) can be used like:

```
>>> from measurement.measures import Distance
>>> distance = Distance(km=10)
```

or can be used for converting other measures into kilometers:

```
>>> from measurement.measures import Distance
>>> distance = Distance(mi=10).km
```

but units that are only acceptable as arguments (like the distance measurement term `kilometer`) can *only* be used to create a measurement:

```
>>> from measurement.measures import Distance
>>> distance = Distance(kilometer=10)
```

You also might notice that some measures have arguments having spaces in their name marked as 'Acceptable as Arguments'; their primary use is for when using `measurement.guess`:

```
>>> from measurement.utils import guess
>>> unit = 'U.S. Foot'
>>> value = 10
>>> measurement = guess(value, unit)
>>> print measurement
10.0 U.S. Foot
```

3.1 Area

- *Acceptable as Arguments or Attributes*: sq_Em, sq_Gm, sq_Mm, sq_Pm, sq_Tm, sq_Ym, sq_Zm, sq_am, sq_british_chain_benoit, sq_british_chain_sears_truncated, sq_british_chain_sears, sq_british_ft, sq_british_yd, sq_chain_benoit, sq_chain_sears, sq_chain, sq_clarke_ft, sq_clarke_link, sq_cm, sq_dam, sq_dm, sq_fathom, sq_fm, sq_ft, sq_german_m, sq_gold_coast_ft, sq_hm, sq_inch, sq_indian_yd, sq_km, sq_link_benoit, sq_link_sears, sq_link, sq_m, sq_mi, sq_mm, sq_nm_uk, sq_nm, sq_pm, sq_rod, sq_sears_yd, sq_survey_ft, sq_um, sq_yd, sq_ym, sq_zm
- *Acceptable as Arguments*: British chain (Benoit 1895 B), British chain (Sears 1922 truncated), British chain (Sears 1922), British foot (Sears 1922), British foot, British yard (Sears 1922), British yard, Chain (Benoit), Chain (Sears), Clarke's Foot, Clarke's link, Foot (International), German legal metre, Gold Coast foot, Indian yard, Link (Benoit), Link (Sears), Nautical Mile (UK), Nautical Mile, U.S. Foot, US survey foot, Yard (Indian), Yard (Sears), attometer, attometre, centimeter, centimetre, decameter, decametre, decimeter, decimetre, exameter, exametre, femtometer, femtometre, foot, gigameter, gigametre, hectometer, hectometre, in, inches, kilometer, kilometre, megameter, megametre, meter, metre, micrometer, micrometre, mile, millimeter, millimetre, nanometer, nanometre, petameter, petametre, picometer, picometre, terameter, terametre, yard, yoctometer, yoctometre, yottameter, yottametre, zeptometer, zeptometre, zetameter, zetametre

3.2 Distance

- *Acceptable as Arguments or Attributes*: Em, Gm, Mm, Pm, Tm, Ym, Zm, am, british_chain_benoit, british_chain_sears_truncated, british_chain_sears, british_ft, british_yd, chain_benoit, chain_sears, chain, clarke_ft, clarke_link, cm, dam, dm, fathom, fm, ft, german_m, gold_coast_ft, hm, inch, indian_yd, km, link_benoit, link_sears, link, m, mi, mm, nm_uk, nm, pm, rod, sears_yd, survey_ft, um, yd, ym, zm
- *Acceptable as Arguments*: British chain (Benoit 1895 B), British chain (Sears 1922 truncated), British chain (Sears 1922), British foot (Sears 1922), British foot, British yard (Sears 1922), British yard, Chain (Benoit), Chain (Sears), Clarke's Foot, Clarke's link, Foot (International), German legal metre, Gold Coast foot, Indian yard, Link (Benoit), Link (Sears), Nautical Mile (UK), Nautical Mile, U.S. Foot, US survey foot, Yard (Indian), Yard (Sears), attometer, attometre, centimeter, centimetre, decameter, decametre, decimeter, decimetre, exameter, exametre, femtometer, femtometre, foot, gigameter, gigametre, hectometer, hectometre, inches, kilometer, kilometre, megameter, megametre, meter, metre, micrometer, micrometre, mile, millimeter, millimetre, nanometer, nanometre, petameter, petametre, picometer, picometre, terameter, terametre, yard, yoctometer, yoctometre, yottameter, yottametre, zeptometer, zeptometre, zetameter, zetametre

3.3 Energy

- *Acceptable as Arguments or Attributes*: C, EJ, Ec, GJ, Gc, J, MJ, Mc, PJ, Pc, TJ, Tc, YJ, Yc, ZJ, Zc, aJ, ac, cJ, c, cc, dJ, daJ, dac, dc, fJ, fc, hJ, hc, kJ, kc, mJ, mc, nJ, nc, pJ, pc, uJ, uc, yJ, yc, zJ, zc
- *Acceptable as Arguments*: Calorie, attocalorie, attojoule, calorie, centicalorie, centijoule, decacalorie, decajoule, decicalorie, decijoule, exacalorie, exajoule,

femtocalorie, femtojoule, gigacalorie, gigajoule, hectocalorie, hectojoule, joule, kilocalorie, kilojoule, megacalorie, megajoule, microcalorie, microjoule, millicalorie, millijoule, nanocalorie, nanojoule, petacalorie, petajoule, picocalorie, picojoule, teracalorie, terajoule, yoctocalorie, yoctojoule, yottacalorie, yottajoule, zeptocalorie, zeptojoule, zetacalorie, zetajoule

3.4 Speed

Note: This is a bi-dimensional measurement; bi-dimensional measures are created by finding an appropriate unit in the measure's primary measurement class, and an appropriate in the measure's reference class, and using them as a double-underscore-separated keyword argument (or, if converting to another unit, as an attribute).

For example, to create an object representing 24 miles-per hour:

```
>>> from measurement.measure import Speed
>>> my_speed = Speed(mile__hour=24)
>>> print my_speed
24.0 mi/hr
>>> print my_speed.km__hr
38.624256
```

- *Primary Measurement*: Distance
- *Reference Measurement*: Time

3.5 Temperature

- *Acceptable as Arguments or Attributes*: `c`, `f`, `k`
- *Acceptable as Arguments*: `celsius`, `fahrenheit`, `kelvin`

Warning: Be aware that, unlike other measures, the zero points of the Celsius and Farenheit scales are arbitrary and non-zero.

If you attempt, for example, to calculate the average of a series of temperatures using `sum`, be sure to supply your 'start' (zero) value as zero Kelvin (read: absolute zero) rather than zero degrees Celsius (which is rather warm comparatively):

```
>>> temperatures = [Temperature(c=10), Temperature(c=20)]
>>> average = sum(temperatures, Temperature(k=0)) / len(temperatures)
>>> print average  # The value will be shown in Kelvin by default since that is the␣
↪starting unit
288.15 k
>>> print average.c  # But, you can easily get the Celsius value
15.0
>>> average.unit = 'c'  # Or, make the measurement report its value in Celsius by␣
↪default
>>> print average
15.0 c
```

3.6 Time

- *Acceptable as Arguments or Attributes*: Esec, Gsec, Msec, Psec, Tsec, Ysec, Zsec, asec, csec, dasec, day, dsec, fsec, hr, hsec, ksec, min, msec, nsec, psec, sec, usec, ysec, zsec
- *Acceptable as Arguments*: attosecond, centisecond, day, decasecond, decisecond, exasecond, femtosecond, gigasecond, hectosecond, hour, kilosecond, megasecond, microsecond, millisecond, minute, nanosecond, petasecond, picosecond, second, terasecond, yoctosecond, yottasecond, zeptosecond, zetasecond

3.7 Volume

- *Acceptable as Arguments or Attributes*: El, Gl, Ml, Pl, Tl, Yl, Zl, al, cl, cubic_centimeter, cubic_foot, cubic_inch, cubic_meter, dal, dl, fl, hl, imperial_g, imperial_oz, imperial_pint, imperial_qt, imperial_tbsp, imperial_tsp, kl, l, ml, nl, pl, ul, us_cup, us_g, us_oz, us_pint, us_qt, us_tbsp, us_tsp, yl, zl
- *Acceptable as Arguments*: Imperial Gram, Imperial Ounce, Imperial Pint, Imperial Quart, Imperial Tablespoon, Imperial Teaspoon, US Cup, US Fluid Ounce, US Gallon, US Ounce, US Pint, US Quart, US Tablespoon, US Teaspoon, attoliter, attolitre, centiliter, centilitre, cubic centimeter, cubic foot, cubic inch, cubic meter, decaliter, decalitre, deciliter, decilitre, exaliter, exalitre, femtoliter, femtolitre, gigaliter, gigalitre, hectoliter, hectolitre, kiloliter, kilolitre, liter, litre, megaliter, megalitre, microliter, microlitre, milliliter, millilitre, nanoliter, nanolitre, petaliter, petalitre, picoliter, picolitre, teraliter, teralitre, yoctoliter, yoctolitre, yottaliter, yottalitre, zeptoliter, zeptolitre, zetaliter, zetalitre

3.8 Weight

- *Acceptable as Arguments or Attributes*: Eg, Gg, Mg, Pg, Tg, Yg, Zg, ag, cg, dag, dg, fg, g, hg, kg, lb, long_ton, mg, ng, oz, pg, short_ton, stone, tonne, ug, yg, zg
- *Acceptable as Arguments*: attogram, centigram, decagram, decigram, exagram, femtogram, gigagram, gram, hectogram, kilogram, long ton, mcg, megagram, metric ton, metric tonne, microgram, milligram, nanogram, ounce, petagram, picogram, pound, short ton, teragram, ton, yoctogram, yottagram, zeptogram, zetagram

CHAPTER 4

Using Measurement Objects

You can import any of the above measures from *measurement.measures* and use it for easily handling measurements like so:

```
>>> from measurement.measures import Weight
>>> w = Weight(lb=135) # Represents 135lbs
>>> print w
135.0 lb
>>> print w.kg
61.234919999999995
```

You can create a measurement unit using any compatible unit and can transform it into any compatible unit. See *Measures* for information about which units are supported by which measures.

To access the raw integer value of a measurement in the unit it was defined in, you can use the 'value' property:

```
>>> print w.value
135.0
```

CHAPTER 5

Guessing Measurements

If you happen to be in a situation where you are processing a list of value/unit pairs (like you might find at the beginning of a recipe), you can use the *guess* function to give you a measurement object.:

```
>>> from measurement.utils import guess
>>> m = guess(10, 'mg')
>>> print repr(m)
Weight(mg=10.0)
```

By default, this will check all built-in measures, and will return the first measure having an appropriate unit. You may want to constrain the list of measures checked (or your own measurement classes, too) to make sure that your measurement is not mis-guessed, and you can do that by specifying the `measures` keyword argument:

```
>>> from measurement.measures import Distance, Temperature, Volume
>>> m = guess(24, 'f', measures=[Distance, Volume, Temperature])
>>> print repr(m)
Temperature(f=24)
```

Warning: It is absolutely possible for this to misguess due to common measurement abbreviations overlapping – for example, both Temperature and Energy accept the argument `c` for representing degrees celsius and calories respectively. It is advisible that you constrain the list of measurements to check to ones that you would consider appropriate for your input data.

If no match is found, a `ValueError` exception will be raised:

```
>>> m = guess(24, 'f', measures=[Distance, Volume])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "measurement/utils.py", line 61, in guess
    ', '.join([m.__name__ for m in measures])
ValueError: No valid measure found for 24 f; checked Distance, Volume
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search