UNUM,

NUMBERS WITH UNITS IN PYTHON

Pierre X. Denis

Spacebel, I. Vandammestraat 5-7, 1560 Hoeilaart - BELGIUM
Phone: ++32 2 658 20 14, Fax: ++32 2 658 20 90
e-mail: pierre.denis@spacebel.be

ABSTRACT

The mainstream programming languages do not consider the units of measurement associated to the numbers they process. In particular, the issue of unit consistency is not addressed by standard type checking. This may cause major software failures that are difficult to track. The present paper firstly summarises the problem of missing units representation. Then, a set of general requirements is proposed. As a proof-of-concept, the Unum module is presented; it is a generic solution developed in the Python programming language. Unum allows to easily associate units to numbers; it performs unit derivation and consistency checking in a dynamic way. As shown in some examples, it may be used for interactive calculations or small applications.

1. INTRODUCTION

The mainstream programming languages and computing devices are natively designed to process numbers. However, our world is made of *quantities*, which are, basically, numbers associated to units of measurement, such as 10 meters, 50 miles-per-hour and 220 volts. The quantities may be combined together in formula, resulting in new quantities. The consistency of units is essential to guarantee the result's validity; this is a basic cornerstone of all scientific and engineering activities (what we mean by 'consistency' will be clarified later). Unfortunately, this consistency issue is not natively covered by programming languages like C++, Java and Ada. Several strategies exist to address the problem, but they are usually implicit, not standardized, burdensome or fuzzy. These shortcomings may cause bugs that may result in disastrous results, such as the loss of NASA's 125 million $ Mars Climate Orbiter (MCO) in September 1999 [1].

The present paper begins by defining a succinct problem statement of 'quantity computation', starting from the shortcomings of current systems and languages. From this, a set of general requirements is proposed. In the following sections, the Unum module is presented ('Unum' stands for Unit on NUMbers); it provides an operational solution developed in the Python language. This proof-of-concept development emphasises generality, integrity and ease-of-use, in the limited scope of dynamic unit inference and consistency checking. Therefore, due to its dynamic nature, Unum mainly targets the domain of interactive calculations in scientific or engineering fields; it can be used also for small applications, provided that performance is not essential.

Several authors have made research in the field, mainly for static unit inference and consistency checking (this is used to be referred as 'dimension types'). Recent works describe solutions in the ML language ([1], [2], [3]), and in Java ([4]). However, these developments do not seem to have been applied in industrial or large-scale applications.

---

[1] The MCO Mishap Investigation Board concluded *"(...) that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file"*. More precisely the Board highlighted a unit discrepancy in the calculation of the thruster impulse bit : the on-board software was using Newton.seconds (N-s) while the ground software was using pounds-seconds (lbf-s); the two units are actually different by a factor 4.45.

2. PROBLEM STATEMENT

2.1 Quantity, Dimension and Units

To avoid confusions, a couple of definitions are provided hereafter[2], in relative accordance to the terminology found in recent works ([1], [2], [3], [4]).

- A *dimension* is a property ascribed to phenomena, bodies, or substances that can be quantified for, or assigned to, a particular phenomenon, body, or substance. Examples are mass and electric charge.

- A *quantity* is a quantifiable or assignable property ascribed to a particular phenomenon, body, or substance. Examples are the mass of the moon and the electric charge of the proton.

- A *unit* is a particular quantity, defined and adopted by convention, with which other particular quantities of the same kind are compared to express their value. Examples are the kilogram and the Volt.

Usually, several different units are applicable for one given dimension, for convenience or historical reasons; they are related by specific conversion factors[3]. The International System of Units (SI) provides a standardisation of physical units. It is founded on seven *base dimensions* ([4], [5]), which are independent by definition : length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity; SI defines respectively seven *base units* : meter, kilogram, second, ampere, kelvin, mole and candela. From this, *derived dimension* and *derived units* may be defined e.g. the force (mass.length/time^2) is expressed as Newton (kg.m/s^2). Beyond SI, it is natural to define other base dimensions such as information quantity and amount of money, with their associated units, for example, bit and euro. This allows to derive a broad range of useful dimensions and units such as transfer rate (byte per second), duration per orbit, cc of fuel per booster, euro per rocket launch, number of statements by module, etc. Obviously, the list is never ending and the choice depends on the domain and how precisely we want to model the reality.

As stated in [3] and [4], the confusion between dimension and unit seems to have mislead some earlier works. Noteworthy, these concepts are important to clarify the notion of consistency checking in programming languages. In statements involving addition, subtraction, comparison of two quantities[4], the dimension must be the same : we will refer this as *dimension consistency*. If dimension is consistent the units used to express the quantities must be the same or a conversion factor must be applied : this will be referred as *unit consistency*; whether this factor must be explicit or inferred by the system is an important design choice. In the above-mentioned MCO failure, the dimension was consistent (an impulse is defined as force.time) but the units were not, the error being exactly quantified by a missing 4.45 factor.

2.2 Limitation of Static Type Checking

Common static type checking, as found in languages such as C++, Java and Ada, actually addresses different issues than dimension or unit consistency. Even with a good typing strategy, it is feasible to code an arithmetic expression with mismatching units, although it is type-compliant for the compiler and error-free at execution time. For instance, considering that variables *height* and *width* represent lengths in inches, nothing prevents the programmer from calculating the expression *height + height*width*, which erroneously mixes length and surface. More basically, if *distance* represents a length in miles, the expression *height + distance* will probably be accepted (if the typing is loose). The first expression doesn't obey dimension consistency while the second one breaks unit consistency. Currently, without proper tools, modules or language extension, the lack of unit is addressed by the programmer : he must guarantee the consistency of each arithmetic expression vis-à-vis the variable's implicit units. In short, type checking eludes unit checking. Of course, comments and documentation are intended to fill the gap -if accurate-, but the fact that dimensions and units are 'hidden' to the programming language remains error-prone.

---

[2] The definitions presented in this section have been freely adapted by the author from an introduction to SI, given on the site of National Institute for Standard and Technologies (NIST) : `http://physics.nist.gov/cuu/Units/introduction.html`

[3] Some units requires special attentions : degrees Celsius and decibels, for example, do not have the simple scaling property; for the sake of simplicity, these cases are not treated in the present paper.

[4] For statically typed languages, we would have added assignments and arguments passing.

Let us mention another problem : the programmer must translate numbers into unit-aware quantities by adequate output formatting with statements like the following

```
printf("height : %f inches",height);
```

This is also a cause of error since the coherence between the quantity's unit and the hard-coded format string can not be guaranteed.

2.3 Other Issues

These above-mentioned shortcomings may cause bugs that are difficult to track; they are usually detected by cross-reviews of specifications and program code. Such errors may seem too obvious for a careful programmer but these are not so uncommon in large projects involving several teams, maybe working in different countries. From a software engineering point of view, the real issue here is not the potential occurrence of such unit inconsistencies but it is the lack of detection mechanisms that are automated, reliable and easy to use. The problem is perfectly well stated : for ages, scientists check the dimensions involved in their formula and, in some cases, are able to infer new formulas by using dimensional analysis. Several solutions could be -and have been- imagined to address the issue. However their deployment is blocked by operational considerations (which are out of the scope of the present paper) : lack of standards, code readability, overhead in resource consumption, intrusiveness, system integration, interface to existing libraries and… need of a 'cultural shift' for the programmer.

3. REQUIREMENTS PROPOSAL

In order to settle the main requirements for a quantity computation system, we propose the following 8 points :

1. **Ability to create any quantity**
   A quantity will be represented by an object composed of a number with a unit; it is advisable here to be truly generic, i.e. to avoid to narrow the system to a specific set of dimension and units like those admitted in the SI system.

2. **Common arithmetic operations on quantities**,
   This should include the four basic operators, exponentiation and comparisons; this encompasses unit derivation for multiplication, division and exponentiation.

3. **Unit/dimension consistency checking and error reporting**
   As explained before this should be performed in any expression involving addition, subtraction, comparison and exponentiation (also assignments and arguments passing for statically typed languages).

4. **Standard syntax for arithmetic expressions**
   For the programmer point-of-view, it is important to avoid excessive overhead in the code writing; in particular, the statements manipulating quantities must remain unchanged.

5. **Integration with existing numeric types and data structures**
   The numeric part of a quantity could be defined on integer, float, complex, etc; any numeric type available in a language could be used, with no constraint or loss in numerical accuracy; furthermore, quantities may be embedded in lists, matrixes, etc.

6. **Explicit or implicit unit conversion**
   The system is able to derive the right factor to apply for a specified target unit (explicit) or for automated unit consistency between quantities with the same dimension (implicit). An extended requirement could be the automated normalisation : it is basically the process of searching which conversions should be iteratively applied on a given unit in order to obtain the shortest unit expression. For example 1 [Pa.cm^2] could be normalized as 0.0001 [N] by applying 1 [Pa] = 1 [N/m^2] and 1 [cm] = 0.01 [m].

7. **Automatic output formatting**
   It is the responsibility of the quantity object to represent itself as a string, in a standard format. This removes redundancy and risk of errors.

8. **Extensibility**
   We mean here the ability to create any unit, with specific names, representations and conversion rules.

These requirements have different degrees of importance : points 1 to 3 are 'strong', they characterize the minimal system that addresses the afore-mentioned issues; points 4 to 8 are 'added-value' requirements, they are advisable for the sake of readability and programming easiness. We have voluntary left out issues like performance, resource consumption and intrusiveness; these points, while important, are mostly related to real-time and embedded systems which require specific attention, beyond the generality of the mentioned requirements.

4. A SOLUTION : UNUM

Starting from the requirement proposal, we have developed from scratch a small but operational Python module called Unum. It is mainly a proof-of-concept development that emphasises generic designing and ease-of-use.

4.1 The Python Language

Python[5] is an interpreted, object-oriented programming language ([6], [7]). It is often compared to Tcl, Perl, Scheme or Java. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing[6]. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface. There is a rich set of libraries, including mathematical, scientific and Internet domains. The Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Mac… Python is copyrighted but freely usable and distributable, even for commercial use.

4.2 Principles and Design

The Unum module uses the object-orientation and overloading features of Python. It is basically made of a class (Unum) and a database of units (Ubase), which are predefined Unum instances. The user can interact with the system through an interactive calculator (Ucalc) or he may write his own Python application. The architecture is depicted on figure 1.

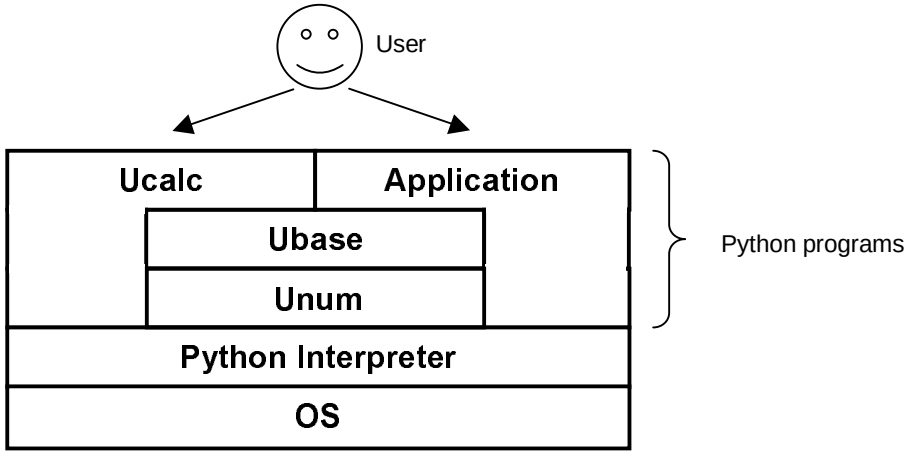

Fig. 1. Unum Architecture

[5] For more information on Python, refer to the official site `http://www.python.org/`.
[6] By 'dynamic typing' we mean that types are associated to objects, not to variables. Concretely, this entails that a given variable may assign successively an integer, a string, a Unum instance, etc.

The Unum class is fully generic : it does not contain any reference to a specific dimension or unit; the set of units, with their representation and conversion rules, is defined in Ubase and may be customized according to the application domain. Each Unum instance (called a *unum*) represent a true quantity, i.e. a number with a unit. Any unum is composed of two private attributes :

- *_value* is the raw numerical value (usually, an integer or a float),
- *_unit* is a dictionary[7] with (string → number) associations : the key is a string identifying a unit and representing it; the associated numeric value is the exponent of that unit.

For instance, the quantity 25 meter/second may be represented by the attributes *_value* = 25 and *_unit* = {'m' : 1, 's' : -1} As a special case, a unum with an empty unit dictionary represents a dimensionless quantity (a raw number). The units defined in Ubase are created as normal Unum instances except that 1° they are referred by a variable name, 2° *_value* is equal to 1 and *_unit* is a singleton with unit representation string associated to an exponent equal to 1, 3° a conversion unum may be provided, which expresses the same quantity in term of more basic unit(s). For instance the unit 'Newton' is represented by a unum with *_value* = 1 and *_unit* = {'N' : 1}, it is associated to a conversion unum defined as *_value* = 1 and *_unit* = {'kg' : 1, 'm' : 1, 's' : -2} (see other examples in following section).

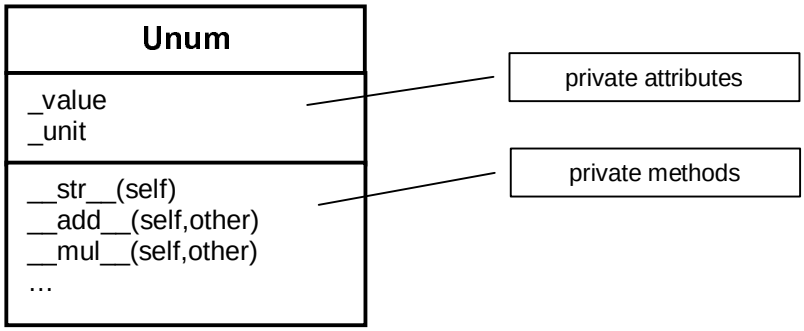

Fig. 2. The Unum Class

Python offers a rich set of overloading methods; for instance the addition of two instances of a given class (*self + other*) can be defined by the programmer by implementing the *__add__(self,other)* method. It is easy then to express arithmetic semantics, as well as coherent unum string representations.

- For multiplication, division, exponentiation, one must derive the exponents of the resulting *_unit* dictionary while removing entries with null exponents. For exponentiation, the exponent must be evaluated as a unit-free unum, otherwise an exception is raised. These operations allow to define any quantity from the predefined 'unit' unums.

- For addition, subtraction and comparison, the dimension consistency requires a matching between the operand's *_unit* dictionaries; if these are exactly equal then the unit consistency is clearly stated. Otherwise, a substitution algorithm will try to unify both units by using the conversion rules; if it succeeds then a conversion factor is applied to one of the operand's *_value*; if it fails then an exception is raised.

In contrast with developments done for statically typed languages ([1], [2], [3], [4]), the concept of dimension does not appear explicitly in this development : we think that such approach limits the flexibility and programming easiness; we argue that it is not mandatory because dimension consistency may be checked by the above-mentioned unification of units.

[7] A dictionary is a data structure that associates keys to values, allowing direct access to values through the keys. Python uses the following syntax to define a dictionary : *{key1 : value1, key2 : value2, …}*

Mixing raw numeric types and unums in an expression is allowed through the Python's coercion method; this, combined with Python's dynamic typing, allows a de facto integration with built-in numeric types or numerical extensions such as rational numbers and matrixes. To conclude this section, let us mention the useful *as()* method, defined in Unum class : it allows to express a unum in any unit given as argument, provided that the dimension is consistent; for example it may translate a power in Watt to kCal/minute. The algorithm is the same as for the addition operator, so an exception is raised in case of inconsistence. For this reason, the *as()* method can also be used as a dimension checker for variable or argument, in the absence of static type checking.

4.3 Examples

We will give here a couple of examples showing Unum in action. The first group of statements is a sample of Ubase, the module defining the units and their relationships.

```
M     = unit('m')                  # meter
CM    = unit('cm'    ,   .01 * M)  # centimeter
KM    = unit('km'    , 1000. * M)  # kilometer
MILES = unit('miles' , 1.609 * KM) # miles
CC    = unit('cc'    , CM**3)      # cubic centimeter
S     = unit('s')                  # second
H     = unit('h'     , 3600. * S)  # hour
KG    = unit('kg')                 # kilogram
G     = unit('g'     , .001 * KG)  # gram
N     = unit('N'     , KG*M/S**2)  # Newton
P     = unit('Pa'    , N/M**2)     # Pascal
J     = unit('J'     , N*M)        # Joule
```

The two following examples are Ucalc sessions; they demonstrate the main features of Unum through the interactive calculator (the output beginning with a '#' are produced by Unum's exceptions).

```
---- Welcome in Unum Calculator ----
>>> distance = 1240 * KM + 500 * M
>>> distance
1240.5 [km]
>>> distance / M
1240500.0 []
>>> speed = 115 * KM/H
>>> speed
115.0 [km/h]
>>> speed.as(MILES/H)
71.4729645743 [miles/h]
>>> distance / speed + 20 * MIN
11.1202898551 [h]
>>> speed + 2*distance
# UNUM ERROR : [km/h] incompatible with [km]
>>> mass = 500 * KG + 250 * G
>>> c = 300000 * KM/S
>>> energy = mass * c
>>> energy
150075000000.0 [g.km/s]
>>> energy.as(KCAL)
# UNUM ERROR : [g.km/s] incompatible with [kcal]
>>> energy = mass * c**2
>>> energy
4.50225e+016 [km2.g/s2]
>>> energy.as(KCAL)
1.07554945055e+016 [kcal]
```

In the next example we demonstrate the ability to create new units on-the-fly and the use of non-physical units (*BOOSTER*, *EURO* and *FF*), in a rocket booster engineering problem.

```
---- Welcome in Unum Calculator ----
>>> BOOSTER = unit('booster')
>>> BOOSTER
1.0 [booster]
>>> EURO = unit('Euro')
>>> FF = unit('FF', 6.125/40.3399 * EURO)
```

```
>>> L = unit('liter', 1000 * CC)
>>> (M**3).as(L)
1000.0 [liter]
>>> capacity_per_booster = 40 * M**3/BOOSTER
>>> capacity_per_booster.as(CC/BOOSTER)
40000000.0 [cc/booster]
>>> fuel_price = 2.5 * EURO/L
>>> fuel_cost = fuel_price * capacity_per_booster * 2 * BOOSTER
>>> fuel_cost
200000.0 [Euro]
>>> fuel_cost.as(FF)
1317221.22449 [FF]
```

To conclude this section, we show the integration of Unum in a small application calculating, for given lists of distances and masses, the gravitation force between two bodies, with their respective accelerations.

```
# -- Import of units
from ubase import *

# -- Constants
K            = 6.673E-11 * N*M**2/KG**2
earth_mass   = 5.980E24  * KG
g            = 9.81 * M/S**2
c            = 300000 * KM/S
earth_radius = ((K*earth_mass/g)**.5).as(KM)

# -- Input Data
distances = (5*CM, earth_radius, c * 365*24*H)
masses    = (5*G, earth_mass, 1000*earth_mass)

# -- Processing and display
print "G            = %s" % K
print "Earth mass   = %s" % earth_mass
print "Earth radius = %s" % earth_radius
print "distances    = %s" % str(distances)
print "masses       = %s" % str(masses)
print
for m1 in masses:
    for m2 in masses:
        if m1 >= m2:
           for d in distances:
               force = K*m1*m2/d**2
               a1 = force/m1
               a2 = force/m2
               print "m1 = %s, m2 = %s, d = %s" % (m1, m2, d)
               print "f = %s, a1 = %s, a2 = %s\n" \
                     % (force.as(N), a1.as(M/S**2), a2.as(M/S**2))
```

The output of this program is sampled hereafter :

```
G            = 6.673e-011 [N.m2/kg2]
Earth mass   = 5.98e+024 [kg]
Earth radius = 6377.88450862 [km]
distances    = (5.0 [cm], 6377.88450862 [km], 9.4608e+012 [km])
masses       = (5.0 [g], 5.98e+024 [kg], 5.98e+027 [kg])

m1 = 5.0 [g], m2 = 5.0 [g], d = 5.0 [cm]
f = 6.673e-013 [N], a1 = 1.3346e-010 [m/s2], a2 = 1.3346e-010 [m/s2]

m1 = 5.0 [g], m2 = 5.0 [g], d = 6377.88450862 [km]
f = 4.10117056856e-029 [N], a1 = 8.20234113712e-027 [m/s2], a2 = 8.20234113712e-027 [m/s2]

m1 = 5.0 [g], m2 = 5.0 [g], d = 9.4608e+012 [km]
f = 1.86382619077e-047 [N], a1 = 3.72765238154e-045 [m/s2], a2 = 3.72765238154e-045 [m/s2]

m1 = 5.98e+024 [kg], m2 = 5.0 [g], d = 5.0 [cm]
f = 7.980908e+014 [N], a1 = 1.3346e-010 [m/s2], a2 = 1.5961816e+017 [m/s2]
```

A stated before, the *force.as(N)* expression has two purposes : it expresses the output data with friendly units (a force as Newton), and, as prerequisite, it checks that the *force* variable is dimensionally consistent with a force : a unit error in the input data can be indirectly detected here, as well as potential errors in the formula.
We clearly see on this last example the advantages of programming with a true quantity computation system : quantities are defined in any unit consistent with the needed dimension; conversion, output formatting and consistency checking are automatically performed throughout the process. This improves the program expressiveness and reduces the risk of errors. The price to pay, in the case of the present development, is the overhead in resource consumption and the lack of static type checking of Python.

5. CONCLUSIONS

The problem of the lack of unit representation in programming languages is clearly stated; missing dimension and unit consistency checking mechanisms can cause important software failures that are difficult to track. To address the issues, we have proposed a set of general requirements and, as a proof-of-concepts, these have been realised in a Python module called Unum. Unum allows to easily express true quantities by extending numbers with units of measurements; it features automatic output formatting, dynamic unit derivation and consistency checking. The Python language has proved to be extremely effective for the Unum implementation, chiefly through the use of dictionaries and operator overloading. Furthermore the language allows a seamless integration with built-in numeric types. The dynamic nature of Unum makes it interesting in the domain of interactive calculations or 'small' applications, as needed in scientific or engineering activities. For domains like embedded or mission-critical software, static unit consistency checking is required in order to avoid overhead in resource consumption. Nevertheless the requirements and design ideas that were presented here could inspire ad hoc extensions of languages with static typing.

Additional information on Unum, including a free download version, are available on
`http://gallery.uunet.be/Pierre-et-Liliane.DENIS/Unum.html`.

6. REFERENCES

1. Andrew Kennedy, 'Dimension Types', *European Symposium on Programming '94*, LNCS Volume 788.

2. Andrew J. Kennedy, 'Relational Parametricity and Units of Measure', *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

3. Jean Goubault, 'Inférence d'unités physiques en ML', *Journées francophones des langages applicatifs'*, INRIA, Noirmoutier, 3 – 20, 1994.

4. André Van Delft, 'A Java Extension with Support for Dimensions', *Software Practice and Experience*, Vol. 29 (7), 605 – 616, 1999.

5. Barry N. Taylor, 'Guide for the Use of the International System of Units (SI)', *NIST Special Publication 811*, 1995.

6. Guido van Rossum, 'Python Reference Manual, Release 1.5.2', CNRI, April 13, 1999

7. Mark Lutz, David Ascher, *Learning Python*, O'Reilly & Associates, 1999