User Guide for an Efficient Scalar Class in Python

Russell A. Paielli*

Abstract— A Python class was designed to represent physical scalars and to eliminate errors involving implicit physical units (e.g., confusing angular degrees and radians). The standard arithmetic operators are overloaded to provide syntax identical to that for built-in numeric types. The scalar class comes with a complete implementation of the standard metric system of units and many common non-metric units. It also allows the user to easily define a specialized or reduced set of appropriate physical units for any particular application or domain. Once an application has been developed and tested, the units can easily be switched off, if desired, to achieve the execution efficiency of operations on built-in numeric types (which can be two orders of magnitude faster). The scalar class can also be used for discrete units to enforce type checking of integer counts, thereby enhancing the built-in dynamic type checking of Python.

Introduction

Physical units and scalars are fundamental to virtually all scientific and engineering calculations. All scientists and engineers learn to add, subtract, multiply, and divide units and to keep track of them with pencil and paper. When they program a computer, however, they usually drop the explicit units and leave them implicit in the actual numerical calculations, perhaps with a comment to document the units. They do that for two basic reasons: (1) software that automatically tracks and checks units, although available, is not widely standardized and well known, and (2) that software can, in some cases, drastically reduce computational efficiency. The objective of the scalar class is to address those problems.

A consequence of the lack of explicit units in most scientific and engineering software is that mistaken units are a common source of error. Perhaps the most common such error involves passing an angle in degrees to a trigonometric function that expects it in radians. Humans tend to prefer degrees, but standard trig functions take radians, and the conversion is often forgotten until its omission is discovered after time-consuming debugging. In one extreme case, simulation results over a period of six months were corrupted by such an error. Many other unit errors also occur, of course. In air traffic management (ATM), for example, horizontal distance is usually specified in nautical miles, whereas altitude is specified in feet, hundreds of feet ("Flight Levels"), or thousands of feet. Those units can easily get confused if comments in the code are the only mechanism for checking consistency.

The approach taken here to prevent such confusion is to allow the user to select units that are appropriate for the job, then to track those units explicitly in software just as an engineer or scientist would do on paper. The scalar class itself does not specify any units, but a unit definition file can be used to define the preferred units and the relevant conversion factors for the particular application or domain. Two such definition files are included with the scalar package, one for metric units and one for traditional ATM units. These files can be extended easily and also serve as simple examples for creating other sets of units.

When run in the interactive Python interpreter, the scalar class can serve as a "calculator" that tracks units and checks consistency. No longer must engineers enter only the numbers into a calculator and manipulate the units separately in their heads or on paper. But the larger benefit of the scalar class is in automatically catching unit errors in Python scripts. For computationally intensive applications, the units can be easily switched off for efficient production runs. Thus, the scalar class can be used during development and testing to guarantee unit correctness, then turned off to obtain the execution efficiency associated with built-in

*Russ Paielli can be reached through his website at http://RussP.us. This document was last updated on 2006-05-26 and applies to version 1.0 of the scalar class, which is available at http://RussP.us/scalar.htm.

numeric types. The resulting improvement in efficiency can be two orders of magnitude (roughly a factor of 100).

Several other software packages are also currently available in Python for representing and manipulating physical scalars with units, but this scalar class was developed independently. Any resemblance to other packages is coincidental. The scalar class presented here is also believed to be the first to provide the option to easily switch off units to realize the efficiency of operations on built-in numeric types. That capability could be a key to widespread adoption, because the large computational overhead involved with tracking and checking units is no longer a reason to avoid using them.

Installation

To install the scalar package, the "tar" file must first be uncompressed and extracted:

```
% gtar -xzvf scalar-x.x.tar.gz
% cd scalar-x.x
```

where "%" is the command prompt, and "`x.x`" is the version number. Since no compilation is necessary, the user can then simply go to the "`scalar`" directory and experiment with the package, if desired. A default installation can be performed by typing

```
% python setup.py install
```

The default installation may require "root" privileges. For more information and other installation options, try

```
% python setup.py --help
```

and

```
% python setup.py install --help
```

Another way to install the scalar class for an individual user is to simply set the "`PYTHONPATH`" environment variable to point to the directory containing the scalar module (i.e., the "`scalar.py`" file). No compilation is necessary. In the standard BASH shell, for example, this can be done by putting the following line in the user's "`.bashrc`" initialization file:

```
export PYTHONPATH=~/scalar-x.x/scalar
```

where the directory would be replaced with the actual location of the scalar module ("`scalar.py`"), of course.

Usage

The scalar class is actually called "`_scalar`" because is not intended to be used directly. Instead, a function called "`unit`" (which calls the `_scalar` class constructor) is intended to be used to define units. But the user need not even call the "`unit`" function directly unless a new unit is needed that is not already available in the unit definition file included with the package. The `units` module (file "`units.py`") defines a comprehensive set of standard units, including the complete SI (metric) system and many common non-metric units. To access those units, simply "import" the `units` module:

```
from units import *
```

The "`units`" module imports the `scalar` module, hence the user should *not* import the `scalar` module directly. Doing so could cause problems if units are later disabled for efficiency, which will be discussed later.

The pre-defined units in the `units` module are as consistent as possible with standard unit abbreviations such as "`s`" for seconds and "`m`" for meters. Thus, for example, the scalar "23 m/s^2" would be "constructed" with

```
vel = 23 * m/s**2
```

These short identifiers are convenient for interactive "calculator" sessions or small applications, but they may be inappropriate for larger applications because short identifiers at global scope can cause problems with name clashes and inadvertent shadowing or overwriting of unit names. To avoid such problems, the `units` module can be imported with

```
import units
```

The meter object would then be referenced as "`units.m`." A cleaner form can be realized with

```
import units as u
```

The meter object is then referenced simply as "`u.m`," which saves on both typing and code clutter, while maintaining the separate namespace.

Another way to avoid the short identifiers at global scope is to use long unit names, such as "`meters`," "`centimeters`," "`seconds`," "`milliseconds`," "`microseconds`," etc., which are also provided. See the `units` module for the complete set of pre-defined unit names.

In addition to the `units` module, which is fairly comprehensive, another smaller module called "`ATMunits`" is also included. It was designed for basic air traffic management (not actual operational ATM, but prototyping, testing, and data analysis). It provides an example of a simplified unit definition file for a particular application or domain without the overhead of all the unused units in the `units` module. Users can also copy the `units` module and strip out what they don't need, of course.

As in the SI system of units, the base unit for length in the `units` module is the meter, and several common scaled variations of it are defined:

```
m  = unit ("m") # meter: length
mm = unit ("mm", m/1000.) # millimeter
um = unit ("um", mm/1000.) # micrometer
cm = unit ("cm", m/100.) # centimeter
km = unit ("km", 1000*m ) # kilometer
```

If only one argument is passed to the unit function, it creates a new base unit, but if two arguments are passed to it, it creates a derived unit that is defined by the second argument. Length outputs will be printed in meters by default unless the user specifies otherwise. For example:

```
>>> dist = 5.2 * m
>>> dist += 27 * mm
>>> print dist
5.227 m
>>> print format(dist,"ft","%2.2f")
17.15 ft
```

The third argument for numeric formatting is optional.

The "`format`" function raises an exception if the unit specified in the second argument is not of the correct type for the scalar object passed in the first argument. The available output units are the keys of the "`to`" dictionary in the `units` module. The "`format`" function is not as convenient as a simple print statement, but its use is highly encouraged for non-trivial programs because it guarantees that the output will not change if units are switched off for efficiency (which will be discussed below). For that reason, it should be used (for non-trivial programs) even when the desired unit is the base unit.

The user is free to define any set of units, but unit names should contain only alphabetic characters (lowercase and/or uppercase letters). Note however, that unit names with non-alphabetic characters will not raise an exception until they are printed out. [This allows output conversions to units with non-alphabetic characters, such as `"ft-lbf"`.]

The convention for printing units is to show multiplication with dashes, division with slashes, and powers with the carat symbol. For example, "kilogram-meters/second-squared" would be shown as "`kg-m/s^2`." Only one slash is allowed, and if the denominator has multiplied units they are placed in parentheses and connected with dashes, such as "`kg/(m-s^2)`."

No function is provided to extract the numerical coefficient of a scalar independent of the units because that would depend on the base units chosen. To write numerical data to a file without the units, or to pass data to a third-party function or application without the units, simply divide the scalar by the required unit. Suppose, for example, that a distance needs to be written or passed as a number with implicit units of feet. If the variable is named "`dist`", then "`dist/ft`" will be automatically cast to a built-in type (typically a `float`) corresponding to the distance in feet.

A scalar cannot be cast to a built-in numeric type such as "`float`" unless it is dimensionless. The predefined units of "`rad`" (radians) and "`deg`" (degrees) in the `units` module are dimensionless, for example, and the base unit is radians. This convention guarantees that standard trigonometric functions will always be passed arguments in terms of radians, as expected, and it prevents any other unit from being erroneously passed to a trigonometric function. For example, `sin(30*deg)` returns 0.5 as expected rather than being converted to `sin(30)`, which would be wrong.

To facilitate the switching off of units for efficiency (to be discussed in the next section), the scalar class has no public member functions (other than the overloaded arithmetic operators). The reason is that calls of member functions using standard "dot" notation cannot work on built-in numeric types.

The scalar module imports the standard math module, and it also defines new global functions "`Sqrt`," "`Hypot`," and "`Atan2`." When used on `float` types, these functions are equivalent to their uncapitalized counterparts in the standard math library, but they also work on scalar types.

Disabling Units for Efficiency

The scalar class can be used as a units "calculator" in the interactive Python shell, and it is also computationally efficient enough for many non-interactive applications. However, it may be too slow for some computationally intensive applications. Computational "overhead" can make operations one to two orders of magnitude (roughly 100 times) slower than corresponding operations on built-in numeric types such as "`float`" and "`int`."

The source of that overhead is twofold. First, the character-string manipulation involved with tracking and checking the units obviously takes some time. But just disabling the unit tracking cannot increase the efficiency to the level of built-in types. The mere fact that scalar is a class rather than a built-in type also adds substantial overhead, slowing the execution speed by up to an order of magnitude.

Fortunately, a simple but innovative method has been devised to eliminate both sources of overhead. After an application has been tested and its unit consistency verified, the user can "switch off" the units for production runs by simply replacing `"units.py"` with `"units_off.py"` in the "`import`" line:

```
from units_off import *
```

or

```
import units_off as u
```

Whereas the `units` module imports the file `"scalar.py"`, the `units_off` module imports `"scalar_off.py"`, which replaces the "`unit`" function with a function of the same name that simply returns 1. Thus, the expression "`t=25*s`" is replaced with "`t=25*1`," eliminating the overhead of the units and the scalar class. It leaves an unnecessary multiplication by one, but that should not be a concern unless it occurs in a highly repetitive loop. If it does, the multiplication by the unit `"s"` can perhaps be moved ahead of the loop.

If a program has more than one module that imports the unit definition module, then disabling the units requires that it be replaced with the "off" version in each module. This procedure can be simplified by creating a one-line module that imports the unit definition module, then importing that module in place of the unit definition module. For example, a file called "`units_.py`" could be created containing a single line:

```
from units import *
```

Then all the other files that need to import the `units` module can import `units_` instead:

```
from units_ import *
```

or

```
import units_ as u
```

Then units can be disabled for the entire program by simply changing the single line in "`units_.py`" to

```
from units_off import *
```

As mentioned earlier, the scalar class has no public member functions (other than the overloaded arithmetic operators). The reason is that calls of member functions using standard "dot" notation cannot work on built-in types. To guarantee that the switching off of units is as easy as possible, public member functions are not provided.

Enhanced Dynamic Type Checking

The scalar class can also be used to implement a stronger form of dynamic type checking than is provided by the built-in dynamic type checking in Python. Suppose, for example, that a discrete count of "bars" needs to be maintained and perhaps passed to a function or object. The user can simply define a unit for it and use it like any other unit:

```
bar = unit("bar")
count = 0
...
count += bar
```

If a count of type "`bar`" is then erroneously added to an incompatible type, for example, the error will be detected and flagged immediately, perhaps saving substantial debugging time.