The decorator module

Author:	Michele Simionato
E-mail:	michele.simionato@gmail.com
Version:	2.3.1 (25 July 2008)
Download page:	http://www.phyast.pitt.edu/~micheles/python/decorator-2.3.0.zip
Installation:	easy_install decorator
License:	BSD license

Contents

Introduction

Python 2.4 decorators are an interesting example of why syntactic sugar matters: in principle, their introduction changed nothing, since they do not provide any new functionality which was not already present in the language; in practice, their introduction has significantly changed the way we structure our programs in Python. I believe the change is for the best, and that decorators are a great idea since:

- decorators help reducing boilerplate code;
- decorators help separation of concerns;
- decorators enhance readability and maintenability;
- decorators are very explicit.

Still, as of now, writing custom decorators correctly requires some experience and it is not as easy as it could be. For instance, typical implementations of decorators involve nested functions, and we all know that flat is better than nested.

The aim of the `decorator` module it to simplify the usage of decorators for the average programmer, and to popularize decorators usage giving examples of useful decorators, such as `memoize`, `tracing`, `redirecting_stdout`, `locked`, etc.

The core of this module is a decorator factory called `decorator`. All decorators discussed here are built as simple recipes on top of `decorator`. You may find their source code in the `_main.py` file, which is generated automatically when you run the doctester (included into the decorator package) on this documentation:

```
$ python doctester.py documentation.txt
```

At the same time the doctester runs all the examples contained here as test cases.

Definitions

Technically speaking, any Python object which can be called with one argument can be used as a decorator. However, this definition is somewhat too large to be really useful. It is more convenient to split the generic class of decorators in two groups:

- *signature-preserving* decorators, i.e. callable objects taking a function as input and returning a function *with the same signature* as output;
- *signature-changing* decorators, i.e. decorators that change the signature of their input function, or decorators returning non-callable objects.

Signature-changing decorators have their use: for instance the builtin classes `staticmethod` and `classmethod` are in this group, since they take functions and return descriptor objects which are not functions, nor callables.

However, signature-preserving decorators are more common and easier to reason about; in particular signature-preserving decorators can be composed together whereas other decorators in general cannot (for instance you cannot meaningfully compose a staticmethod with a classmethod or viceversa).

Writing signature-preserving decorators from scratch is not that obvious, especially if one wants to define proper decorators that can accept functions with any signature. A simple example will clarify the issue.

Statement of the problem

Suppose you want to trace a function: this is a typical use case for a decorator and you can find in many places code like this:

```
#<_main.py>

try:
    from functools import update_wrapper
except ImportError: # using Python version < 2.5
    def decorator_trace(f):
        def newf(*args, **kw):
           print "calling %s with args %s, %s" % (f.__name__, args, kw)
           return f(*args, **kw)
        newf.__name__ = f.__name__
        newf.__dict__.update(f.__dict__)
        newf.__doc__ = f.__doc__
        newf.__module__ = f.__module__
        return newf
else: # using Python 2.5+
    def decorator_trace(f):
        def newf(*args, **kw):
            print "calling %s with args %s, %s" % (f.__name__, args, kw)
            return f(*args, **kw)
        return update_wrapper(newf, f)

#</_main.py>
```

The implementation above works in the sense that the decorator can accept functions with generic signatures; unfortunately this implementation does *not* define a signature-preserving decorator, since in general `decorator_trace` returns a function with a *different signature* from the original function.

Consider for instance the following case:

```
>>> @decorator_trace
... def f1(x):
...     pass
```

Here the original function takes a single argument named `x`, but the decorated function takes any number of arguments and keyword arguments:

```
>>> from inspect import getargspec
>>> print getargspec(f1)
([], 'args', 'kw', None)
```

This means that introspection tools such as pydoc will give wrong informations about the signature of `f1`. This is pretty bad: pydoc will tell you that the function accepts a generic signature `*args`, `**kw`, but when you try to call the function with more than an argument, you will get an error:

```
>>> f1(0, 1)
Traceback (most recent call last):
   ...
TypeError: f1() takes exactly 1 argument (2 given)
```

The solution

The solution is to provide a generic factory of generators, which hides the complexity of making signature-preserving decorators from the application programmer. The `decorator` factory allows to define decorators without the need to use nested functions or classes. As an example, here is how you can define `decorator_trace`.

First of all, you must import `decorator`:

```
>>> from decorator import decorator
```

Then you must define an helper function with signature `(f, *args, **kw)` which calls the original function `f` with arguments `args` and `kw` and implements the tracing capability:

```
#<_main.py>

def trace(f, *args, **kw):
    print "calling %s with args %s, %s" % (f.func_name, args, kw)
    return f(*args, **kw)

#</_main.py>
```

`decorator` is able to convert the helper function into a signature-preserving decorator object, i.e is a callable object that takes a function and returns a decorated function with the same signature of the original function. Therefore, you can write the following:

```
>>> @decorator(trace)
... def f1(x):
...     pass
```

It is immediate to verify that `f1` works

```
>>> f1(0)
calling f1 with args (0,), {}
```

and it that it has the correct signature:

```
>>> print getargspec(f1)
(['x'], None, None, None)
```

The same decorator works with functions of any signature:

```
>>> @decorator(trace)
... def f(x, y=1, z=2, *args, **kw):
...     pass

>>> f(0, 3)
calling f with args (0, 3, 2), {}

>>> print getargspec(f)
(['x', 'y', 'z'], 'args', 'kw', (1, 2))
```

That includes even functions with exotic signatures like the following:

```
>>> @decorator(trace)
... def exotic_signature((x, y)=(1,2)): return x+y

>>> print getargspec(exotic_signature)
([['x', 'y']], None, None, ((1, 2),))
>>> exotic_signature()
calling exotic_signature with args ((1, 2),), {}
3
```

`decorator` is a decorator

The `decorator` factory itself can be considered as a signature-changing decorator, just as `classmethod` and `staticmethod`. However, `classmethod` and `staticmethod` return generic objects which are not callable, while `decorator` returns signature-preserving decorators, i.e. functions of a single argument. Therefore, you can write

```
>>> @decorator
... def tracing(f, *args, **kw):
...     print "calling %s with args %s, %s" % (f.func_name, args, kw)
...     return f(*args, **kw)
```

and this idiom is actually redefining `tracing` to be a decorator. We can easily check that the signature has changed:

```
>>> print getargspec(tracing)
(['func'], None, None, None)
```

Therefore now `tracing` can be used as a decorator and the following will work:

```
>>> @tracing
... def func(): pass

>>> func()
calling func with args (), {}
```

BTW, you may use the decorator on lambda functions too:

```
>>> tracing(lambda : None)()
calling <lambda> with args (), {}
```

For the rest of this document, I will discuss examples of useful decorators built on top of `decorator`.

memoize

This decorator implements the `memoize` pattern, i.e. it caches the result of a function in a dictionary, so that the next time the function is called with the same input parameters the result is retrieved from the cache and not recomputed. There are many implementations of `memoize` in http://www.python.org/moin/PythonDecoratorLibrary, but they do not preserve the signature.

```
#<_main.py>

from decorator import *

def getattr_(obj, name, default_thunk):
    "Similar to .setdefault in dictionaries."
    try:
        return getattr(obj, name)
    except AttributeError:
        default = default_thunk()
        setattr(obj, name, default)
        return default

@decorator
def memoize(func, *args):
```

```
    dic = getattr_(func, "memoize_dic", dict)
    # memoize_dic is created at the first call
    if args in dic:
        return dic[args]
    else:
        result = func(*args)
        dic[args] = result
        return result

#</_main.py>
```

Here is a test of usage:

```
>>> @memoize
... def heavy_computation():
...     time.sleep(2)
...     return "done"

>>> print heavy_computation() # the first time it will take 2 seconds
done

>>> print heavy_computation() # the second time it will be instantaneous
done
```

As an exercise, try to implement `memoize` *properly* without the `decorator` factory.

For sake of semplicity, my implementation only works for functions with no keyword arguments. One can relax this requirement, and allow keyword arguments in the signature, for instance by using `(args, tuple(kwargs.iteritems()))` as key for the memoize dictionary. Notice that in general it is impossible to memoize correctly something that depends on mutable arguments.

locked

There are good use cases for decorators is in multithreaded programming. For instance, a `locked` decorator can remove the boilerplate for acquiring/releasing locks [1].

```
#<_main.py>

import threading

@decorator
def locked(func, *args, **kw):
    lock = getattr_(func, "lock", threading.Lock)
    lock.acquire()
```

```
    try:
        result = func(*args, **kw)
    finally:
        lock.release()
    return result

#</_main.py>
```

To show an example of usage, suppose one wants to write some data to an external resource which can be accessed by a single user at once (for instance a printer). Then the access to the writing function must be locked:

```
#<_main.py>

import time

datalist = [] # for simplicity the written data are stored into a list.

@locked
def write(data):
    "Writing to a sigle-access resource"
    time.sleep(1)
    datalist.append(data)

#</_main.py>
```

Since the writing function is locked, we are guaranteed that at any given time there is at most one writer. An example multithreaded program that invokes `write` and prints the datalist is shown in the next section.

`delayed` and `threaded`

Often, one wants to define families of decorators, i.e. decorators depending on one or more parameters.

Here I will consider the example of a one-parameter family of `delayed` decorators taking a procedure and converting it into a delayed procedure. In this case the time delay is the parameter.

A delayed procedure is a procedure that, when called, is executed in a separate thread after a certain time delay. The implementation is not difficult:

[1] In Python 2.5, the preferred way to manage locking is via the `with` statement: http://docs.python.org/lib/with-locks.html

```
#<_main.py>

def delayed(nsec):
    def call(proc, *args, **kw):
        thread = threading.Timer(nsec, proc, args, kw)
        thread.start()
        return thread
    return decorator(call)

#</_main.py>
```

Notice that without the help of `decorator`, an additional level of nesting would have been needed.

Delayed decorators as intended to be used on procedures, i.e. on functions returning `None`, since the return value of the original function is discarded by this implementation. The decorated function returns the current execution thread, which can be stored and checked later, for instance to verify that the thread `.isAlive()`.

Delayed procedures can be useful in many situations. For instance, I have used this pattern to start a web browser *after* the web server started, in code such as

```
>>> @delayed(2)
... def start_browser():
...     "code to open an external browser window here"

>>> #start_browser() # will open the browser in 2 seconds
>>> #server.serve_forever() # enter the server mainloop
```

The particular case in which there is no delay is important enough to deserve a name:

```
#<_main.py>

threaded = delayed(0) # no-delay decorator

#</_main.py>
```

Threaded procedures will be executed in a separated thread as soon as they are called. Here is an example using the `write` routine defined before:

```
>>> @threaded
... def writedata(data):
...     write(data)
```

Each call to `writedata` will create a new writer thread, but there will be no synchronization problems since `write` is locked.

```
>>> writedata("data1")
<_Timer(Thread-1, started)>
```

```
>>> time.sleep(.1) # wait a bit, so we are sure data2 is written after data1

>>> writedata("data2")
<_Timer(Thread-2, started)>

>>> time.sleep(2) # wait for the writers to complete

>>> print datalist
['data1', 'data2']
```

blocking

Sometimes one has to deal with blocking resources, such as `stdin`, and sometimes it is best to have back a "busy" message than to block everything. This behavior can be implemented with a suitable decorator:

```
#<_main.py>

def blocking(not_avail="Not Available"):
    def call(f, *args, **kw):
        if not hasattr(f, "thread"): # no thread running
            def set_result(): f.result = f(*args, **kw)
            f.thread = threading.Thread(None, set_result)
            f.thread.start()
            return not_avail
        elif f.thread.isAlive():
            return not_avail
        else: # the thread is ended, return the stored result
            del f.thread
            return f.result
    return decorator(call)

#</_main.py>
```

Functions decorated with `blocking` will return a busy message if the resource is unavailable, and the intended result if the resource is available. For instance:

```
>>> @blocking("Please wait ...")
... def read_data():
...     time.sleep(3) # simulate a blocking resource
...     return "some data"

>>> print read_data() # data is not available yet
Please wait ...
```

```
>>> time.sleep(1)
>>> print read_data() # data is not available yet
Please wait ...

>>> time.sleep(1)
>>> print read_data() # data is not available yet
Please wait ...

>>> time.sleep(1.1) # after 3.1 seconds, data is available
>>> print read_data()
some data
```

redirecting_stdout

Decorators help in removing the boilerplate associated to `try .. finally` blocks. We saw the case of `locked`; here is another example:

```
#<_main.py>

import sys

def redirecting_stdout(new_stdout):
    def call(func, *args, **kw):
        save_stdout = sys.stdout
        sys.stdout = new_stdout
        try:
            result = func(*args, **kw)
        finally:
            sys.stdout = save_stdout
        return result
    return decorator(call)

#</_main.py>
```

Here is an example of usage:

```
>>> from StringIO import StringIO

>>> out = StringIO()

>>> @redirecting_stdout(out)
... def helloworld():
...     print "hello, world!"
```

```
>>> helloworld()

>>> out.getvalue()
'hello, world!\n'
```

Similar tricks can be used to remove the boilerplate associate with transactional databases. I think you got the idea, so I will leave the transactional example as an exercise for the reader. Of course in Python 2.5 these use cases can also be addressed with the `with` statement.

Class decorators and decorator factories

Starting from Python 2.6 it is possible to decorate classes. The decorator module takes advantage of this feature to provide a facility for writing complex decorator factories. We have already seen examples of simple decorator factories, implemented as functions returning a decorator. For more complex situations, it is more convenient to implement decorator factories as classes returning callable objects that can be used as signature-preserving decorators. To this aim, `decorator` can also be used as a class decorator. Given a class with a `.call(self, func, *args, **kw)` method `decorator(cls)` adds a suitable `__call__` method to the class; it raises a TypeError if the class already has a nontrivial `__call__` method.

To give an example of usage, let me show a (simplistic) permission system based on classes. Suppose we have a (Web) framework with the following user classes:

```
#<_main.py>

   class User(object):
       "Will just be able to see a page"

   class PowerUser(User):
       "Will be able to add new pages too"

   class Admin(PowerUser):
       "Will be able to delete pages too"

#</_main.py>
```

Suppose we have a function `get_userclass` returning the class of the user logged in our system: in a Web framework `get_userclass` will read the current user from the environment (i.e. from REMOTE_USER) and will compare it with a database table to determine her user class. For the sake of the example, let us use a trivial function:

```
#<_main.py>
```

```
   def get_userclass():
       return User

#</_main.py>
```

We can implement the `Restricted` decorator factory as follows:

```
#<_main.py>

   class PermissionError(Exception):
       pass

   #@decorator # remove the comment for Python >= 2.6
   class Restricted(object):
       """
       Restrict public methods and functions to a given class of users.
       If instantiated twice with the same userclass return the same
       object.
       """
       _cache = {}
       def __new__(cls, userclass):
           if userclass in cls._cache:
               return cls._cache[userclass]
           self = cls._cache[userclass] = super(Restricted, cls).__new__(cls)
           self.userclass = userclass
           return self
       def call(self, func, *args, **kw):
           userclass = get_userclass()
           if issubclass(userclass, self.userclass):
               return func(*args, **kw)
           else:
               raise PermissionError(
               '%s does not have the permission to run %s!'
               % (userclass.__name__, func.__name__))

   Restricted = decorator(Restricted) # use this for Python < 2.6

#</_main.py>
```

An user can perform different actions according to her class:

```
#<_main.py>

   class Action(object):
```

```
        @Restricted(User)
        def view(self):
            pass

        @Restricted(PowerUser)
        def insert(self):
            pass

        @Restricted(Admin)
        def delete(self):
            pass

#</_main.py>
```

Here is an example of usage:

```
>>> a = Action()
>>> a.view()
>>> a.insert()
Traceback (most recent call last):
  ...
PermissionError: User does not have the permission to run insert!
>>> a.delete()
Traceback (most recent call last):
  ...
PermissionError: User does not have the permission to run delete!
```

A `PowerUser` could call `.insert` but not `.delete`, whereas and `Admin` can call all the methods.

I could have provided the same functionality by means of a mixin class (say `DecoratorMixin`) providing a `__call__` method. Within that design an user should have derived his decorator class from `DecoratorMixin`. However, I generally dislike inheritance and I do not want to force my users to inherit from a class of my choice. Using the class decorator approach my user is free to use any class she wants, inheriting from any class she wants, provided the class provide a proper `.call` method and does not provide a custom `__call__` method. In other words, I am trading (less stringent) interface requirements for (more stringent) inheritance requirements.

Dealing with third party decorators: `new_wrapper`

Sometimes you find on the net some cool decorator that you would like to include in your code. However, more often than not the cool decorator is not signature-preserving. Therefore you may want an easy way to upgrade third party decorators to signature-preserving decorators without having to rewrite them in terms of `decorator`. To this aim

the `decorator` module provides an utility function called `new_wrapper`. `new_wrapper` takes a wrapper function with a generic signature and returns a copy of it with the right signature. For instance, suppose you have a wrapper function `wrapper` (or generically a callable object) with a "permissive" signature (say `wrapper(*args, **kw)`) returned by a third party non signature-preserving decorator; let `model` be the original function, with a stricter signature; then `new_wrapper(wrapper, model)` returns a copy of `wrapper` with signature copied from `model`. Notice that it is your responsability to make sure that the original function and the model function have compatibile signature, i.e. that the signature of the model is stricter (or equivalent) than the signature of the original function. If not, you will get an error at calling time, not at decoration time.

With `new_wrapper` at your disposal, it is a breeze to define an utility to upgrade old-style decorators to signature-preserving decorators:

```
#<_main.py>

def upgrade_dec(dec):
    return lambda f : new_wrapper(dec(f), f)

#</_main.py>
```

tail_recursive

In order to give an example of usage for `new_wrapper`, I will show a pretty slick decorator that converts a tail-recursive function in an iterative function. I have shamelessly stolen the basic idea from Kay Schluehr's recipe in the Python Cookbook, http://aspn.activestate.com/ASPN/

```
#<_main.py>

from decorator import new_wrapper

class TailRecursive(object):
    """
    tail_recursive decorator based on Kay Schluehr's recipe
    http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/496691
    """
    CONTINUE = object() # sentinel

    def __init__(self, func):
        self.func = func
        self.firstcall = True

    def __call__(self, *args, **kwd):
        try:
```

```
            if self.firstcall: # start looping
                self.firstcall = False
                while True:
                    result = self.func(*args, **kwd)
                    if result is self.CONTINUE: # update arguments
                        args, kwd = self.argskwd
                    else: # last call
                        break
            else: # return the arguments of the tail call
                self.argskwd = args, kwd
                return self.CONTINUE
        except: # reset and re-raise
            self.firstcall = True
            raise
        else: # reset and exit
            self.firstcall = True
            return result

tail_recursive = upgrade_dec(TailRecursive)

#</_main.py>
```

Here the decorator is implemented as a class returning callable objects. `upgrade_dec` converts that class in a factory function returning functions. Here is how you apply the upgraded decorator to the good old factorial:

```
#<_main.py>

@tail_recursive
def factorial(n, acc=1):
    "The good old factorial"
    if n == 0: return acc
    return factorial(n-1, n*acc)

#</_main.py>

>>> print factorial(4)
24
```

This decorator is pretty impressive, and should give you some food for your mind ;) Notice that there is no recursion limit now, and you can easily compute `factorial(1001)` or larger without filling the stack frame. Notice also that the decorator will not work on functions which are not tail recursive, such as

```
def fact(n): # this is not tail-recursive
    if n == 0: return 1
    return n * fact(n-1)
```

(a function is tail recursive if it either returns a value without making a recursive call, or returns directly the result of a recursive call).

Caveats and limitations

The first thing you should be aware of, it the fact that decorators have a performance penalty. The worse case is shown by the following example:

```
$ cat performance.sh
python -m timeit -s "
from decorator import decorator

@decorator
def do_nothing(func, *args, **kw):
    return func(*args, **kw)

@do_nothing
def f():
    pass
" "f()"

python -m timeit -s "
def f():
    pass
" "f()"
```

On my Linux system, using the `do_nothing` decorator instead of the plain function is more than four times slower:

```
$ bash performance.sh
1000000 loops, best of 3: 1.68 usec per loop
1000000 loops, best of 3: 0.397 usec per loop
```

It should be noted that a real life function would probably do something more useful than `f` here, and therefore in real life the performance penalty could be completely negligible. As always, the only way to know if there is a penalty in your specific use case is to measure it.

You should be aware that decorators will make your tracebacks longer and more difficult to understand. Consider this example:

```
>>> @tracing
... def f():
...     1/0
```

Calling `f()` will give you a `ZeroDivisionError`, but since the function is decorated the traceback will be longer:

```
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    f()
  File "<string>", line 2, in f
  File "<stdin>", line 4, in tracing
    return f(*args, **kw)
  File "<stdin>", line 3, in f
    1/0
ZeroDivisionError: integer division or modulo by zero
```

You see here the inner call to the decorator `tracing`, which calls `f(*args, **kw)`, and a reference to `File "<string>", line 2, in f`. This latter reference is due to the fact that internally the decorator module uses `eval` to generate the decorated function. Notice that `eval` is *not* responsibile for the performance penalty, since is the called *only once* at function decoration time, and not every time the decorated function is called.

Using `eval` means that `inspect.getsource` will not work for decorated functions. This means that the usual '??' trick in IPython will give you the (right on the spot) message `Dynamically generated function. No source code available.`. This however is preferable to the situation with regular decorators, where `inspect.getsource` gives you the wrapper source code which is probably not what you want:

```
#<_main.py>

def identity_dec(func):
    def wrapper(*args, **kw):
        return func(*args, **kw)
    return wrapper

@identity_dec
def example(): pass

#</_main.py>
```

```
>>> import inspect
>>> print inspect.getsource(example)
    def wrapper(*args, **kw):
        return func(*args, **kw)
<BLANKLINE>
```

(see bug report 1764286 for an explanation of what is happening).

At present, there is no clean way to avoid `eval`. A clean solution would require to change the CPython implementation of functions and add an hook to make it possible to change their signature directly. This will happen in future versions of Python (see PEP 362) and then the decorator module will become obsolete.

For debugging purposes, it may be useful to know that the decorator module also provides a `getinfo` utility function which returns a dictionary containing information about a function. For instance, for the factorial function we will get

```
>>> d = getinfo(factorial)
>>> d['name']
'factorial'
>>> d['argnames']
['n', 'acc']
>>> d['signature']
'n, acc'
>>> d['defaults']
(1,)
>>> d['doc']
'The good old factorial'
```

In the present implementation, decorators generated by `decorator` can only be used on user-defined Python functions or methods, not on generic callable objects, nor on built-in functions, due to limitations of the `inspect` module in the standard library. Also, there is a restriction on the names of the arguments: if try to call an argument `_call_` or `_func_` you will get an AssertionError:

```
>>> @tracing
... def f(_func_): print f
...
Traceback (most recent call last):
  ...
AssertionError: You cannot use _call_ or _func_ as argument names!
```

(the existence of these two reserved names is an implementation detail).

Moreover, the implementation is such that the decorated function contains a copy of the original function attributes:

```
>>> def f(): pass # the original function
>>> f.attr1 = "something" # setting an attribute
>>> f.attr2 = "something else" # setting another attribute

>>> traced_f = tracing(f) # the decorated function

>>> traced_f.attr1
'something'
```

```
>>> traced_f.attr2 = "something different" # setting attr
>>> f.attr2 # the original attribute did not change
'something else'
```

That's all folks, enjoy!

LICENCE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

> Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in bytecode form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

If you use this software and you are happy with it, consider sending me a note, just to gratify my ego. On the other hand, if you use this software and you are unhappy with it, send me a patch!