classtools Documentation

Release 0.1

Eevee

May 07, 2017

Contents

Collection of small class-related utilities that will hopefully save you a tiny bit of grief, in the vein of itertools and functools.

class `classtools.`**`classproperty`**(*desc*, *fget*, *fset=None*, *fdel=None*)

Method decorator similar to `@property`, but called like a `classmethod`.

```
class Square(object):
    @classproperty
    def num_sides(cls):
        return 4
```

Setting and deleting are not supported, due to the design of the descriptor protocol. If you need a class property you can set or delete, you need to create a metaclass and put a regular `@property` on it.

This decorator is of questionable use for normal classes, but may be helpful for "declarative" classes such as enums.

class `classtools.`**`frozenproperty`**(*fget*)

Similar to the built-in `@property` decorator, but without support for setters or deleters. This makes it a "non-data" descriptor, which you can overwrite directly. Compare:

```
class Cat(object):
    @property
    def num_legs(self):
        return 2 + 2
```

```
>>> cat = Cat()
>>> cat.num_legs
4
>>> cat.num_legs = 5
...
AttributeError: can't set attribute
```

Versus:

```
class Cat(object):
    @frozenproperty
    def num_legs(self):
        return 2 + 2
```

```
>>> cat = Cat()
>>> cat.num_legs
4
>>> cat.num_legs = 5
>>> cat.num_legs
5
```

`classtools.`**`keyed_ordering`**(*cls*)

Class decorator to generate all six rich comparison methods, based on a `__key__` method.

Many simple classes are wrappers for very simple data, and want to defer comparisons to that data. Rich comparison is very flexible and powerful, but makes this simple case tedious to set up. There's the standard library's `total_ordering` decorator, but it still requires you to write essentially the same method twice, and doesn't correctly handle `NotImplemented` before 3.4. It also doesn't automatically generate `__ne__` from `__eq__`, which is a common gotcha.

With this decorator, comparisons will be done on the return value of `__key__`, in much the same way as the `key` argument to `sorted`. For example, if you have a class representing a span of time:

```
@keyed_ordering
class TimeSpan(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __key__(self):
        return (self.start, self.end)
```

This is equivalent to the following, assuming 3.4's `total_ordering`:

```
@total_ordering
class TimeSpan(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __eq__(self, other):
        if not isinstance(other, TimeSpan):
            return NotImplemented
        return (self.start, self.end) == (other.start, other.end)

    def __ne__(self, other):
        if not isinstance(other, TimeSpan):
            return NotImplemented
        return (self.start, self.end) != (other.start, other.end)

    def __lt__(self, other):
        if not isinstance(other, TimeSpan):
            return NotImplemented
        return (self.start, self.end) < (other.start, other.end)
```

The `NotImplemented` check is based on the class being decorated, so subclassses can still be correctly compared.

You may also implement some of the rich comparison methods in the decorated class, in which case they'll be left alone.

class `classtools.`**`reify`**(*wrapped*)

Method decorator similar to `@property`, except that after the wrapped method is called, its return value is stored in the instance dict, effectively replacing this decorator. The name means "to make real", because some value becomes a "real" instance attribute as soon as it's computed.

The wrapped method is thus only called once at most, making this decorator particularly useful for lazy/delayed creation of resources, or expensive computations that will always have the same result but may not be needed at all. For example:

```
class Resource(object):
    @reify
    def result(self):
        print('fetching result')
        return "foo"
```

```
>>> r = Resource()
>>> r.result
fetching result
'foo'
>>> r.result
```

```
'foo'
>>> # result not called the second time, because r.result is now populated
```

Because this is a "non-data descriptor", it's possible to set or delete the attribute:

```
>>> r.result = "bar"
>>> r.result
'bar'
>>> del r.result
>>> r.result
fetching result
'foo'
```

Deleting the attribute causes the wrapped method to be called again on the next read. While it's possible to take advantage of this to create a cache with manual eviction, the author strongly advises you not to think of this decorator as merely a caching mechanism. Strictly speaking, cache eviction should only ever affect performance, but consider the following:

```
class Database(object):
    @reify
    def connection(self):
        return dbapi.connect(...)
```

Here, `reify` is used as lazy initialization, and its return value is a (mutable!) handle to some external resource. That handle is not cached in any meaningful sense: its permanence is guaranteed by the class. Having it transparently evicted and recreated would not only destroy the illusion that it's a regular attribute, but completely break the class's semantics.

class `classtools.`**`weakattr`**(*name*)

Descriptor that transparently wraps its stored value in a weak reference. Reading this attribute will never raise *AttributeError*; if the reference is broken or missing, you'll just get *None*.

To use, create a `weakattr` in the class body and assign to it as normal. You must provide an attribute name, which is used to store the actual weakref in the instance dict.

```
class Foo(object):
    bar = weakattr('bar')

    def __init__(self, bar):
        self.bar = bar
```

```
>>> class Dummy(object): pass
>>> obj = Dummy()
>>> foo = Foo(obj)
>>> assert foo.bar is obj
>>> print(foo.bar)
<object object at ...>
>>> del obj
>>> print(foo.bar)
None
```

Of course, if you try to assign a value that can't be weak referenced, you'll get a `TypeError`. So don't do that. In particular, a lot of built-in types can't be weakref'd!

Note that due to the `__dict__` twiddling, this descriptor will never trigger `__getattr__`, `__setattr__`, or `__delattr__`.

Python Module Index

C

Index