28-decorators

April 24, 2016

1 Decorators

- Documentation tends to be confusing, but basics are straightforward
- Functions, classes, and methods can be ‘decorated’
- Will only show how to decorate functions - others are fairly complex
- Similar to ‘annotations and aspect’ programming in java
- Good for ‘cross cutting’ concerns, like security, mettering, billing.

2 Callables

- a ‘callable’ is something that can be ‘called’, applied to arguments
- have seen functions and lambdas
- objects can also be callables, by defining the `__call__` method

```
In [2]: import math

        class Co:
            # args applied to object will call this
            def __call__(self, x):
                return(math.sin(x))


        # make a Co object
        c = Co()

        # can call object like a function
        [math.sin(.5), c(.5)]
```

```
Out[2]: [0.479425538604203, 0.479425538604203]
```

```
In [3]: # good old recursive factorial, with a print debug statement added

        def fact(n):
            print(’inside fact({})’.format(n))
            if n == 0:
                return(1)
            else:
                return(n * fact(n-1))

        fact(4)
```

```
inside fact(4)
inside fact(3)
inside fact(2)
inside fact(1)
inside fact(0)
```

```
Out[3]: 24
```

3 to decorate a function, define a class

- can also use nested functions, but a class is easier

```
In [4]: class traceindent(object):
            def __init__(self, f):
                # f is the original function
                self.f = f
                self.level = 0

            def __call__(self, *pos, **kw):
                self.level += 1
                indent = ['.'] * self.level
                indent = ''.join(indent)
                if len(pos) == 1:
                    printpos = '({})'.format(pos[0])
                print("{}Entering {}{}".format(indent, self.f.__name__, printpos))
                # calling the traced function
                val = self.f(*pos, **kw)
                print('{}Exiting {}{}=>{}'.format(indent, self.f.__name__, printpos, val))
                self.level -= 1
                return(val)
```

```
In [7]: # decorate the fact function with a trace

        @traceindent
        def fact(n):
            if n == 0:
                return(1)
            else:
                return(n * fact(n-1))

        fact(4)
```

```
.Entering fact(4)
..Entering fact(3)
...Entering fact(2)
...Entering fact(1)
...Entering fact(0)
...Exiting fact(0)=>1
...Exiting fact(1)=>1
...Exiting fact(2)=>2
..Exiting fact(3)=>6
.Exiting fact(4)=>24
```

```
Out[7]: 24
```

```
In [8]: # 'fact' is an object now, not the original 'def'
        fact
```

```
Out[8]: <__main__.traceindent at 0x105e754e0>
```

4 functools module

- has some decorators
- doc

```
In [ ]: # in the poly class i had to define too many comparison methods
        # here i just do the essentials, and the decorator adds the other methods

        from functools import total_ordering

        @total_ordering
        class Student:
            def __eq__(self, other):
                return ((self.lastname.lower(), self.firstname.lower()) ==
                        (other.lastname.lower(), other.firstname.lower()))
            def __lt__(self, other):
                return ((self.lastname.lower(), self.firstname.lower()) <
                        (other.lastname.lower(), other.firstname.lower()))
```

```
In [9]: # can fill in some args - functional programing types like this

        from functools import partial
        basetwo = partial(int, base=2)
        basetwo.__doc__ = 'Convert base 2 string to an int.'
        basetwo('10010')
```

```
Out[9]: 18
```

```
In [10]: # f[n] = f[n-1] + f[n-2]
         # doubly recursive
         # many redundant calls...

         def fibonacci(n):
            "Return the nth fibonacci number."
            print('in fib', n)
            if n in (0,1):
               return n
            return fibonacci(n-1) + fibonacci(n-2)

         fibonacci(7)
```

```
in fib 7
in fib 6
in fib 5
in fib 4
in fib 3
in fib 2
in fib 1
in fib 0
in fib 1
in fib 2
in fib 1
in fib 0
in fib 3
in fib 2
in fib 1
in fib 0
in fib 1
in fib 4
in fib 3
in fib 2
in fib 1
in fib 0
in fib 1
in fib 2
in fib 1
in fib 0
in fib 5
in fib 4
in fib 3
in fib 2
in fib 1
in fib 0
in fib 1
in fib 2
in fib 1
in fib 0
in fib 3
in fib 2
in fib 1
in fib 0
in fib 1
```

```
Out[10]: 13
```

```
In [11]: import collections
         import functools

         class memoized(object):
             '''Decorator. Caches a function's return value each time it is called.
             If called later with the same arguments, the cached value is returned
             (not reevaluated).
             '''
             def __init__(self, func):
                 self.func = func
                 self.cache = {}

             def __call__(self, *args):
                 if not isinstance(args, collections.Hashable):
                     # uncacheable. a list, for instance.
                     # better to not cache than blow up.
                     return self.func(*args)
                 if args in self.cache:
                     return self.cache[args]
                 else:
                  value = self.func(*args)
                  self.cache[args] = value
                  return value

            def __repr__(self):
               '''Return the function's docstring.'''
               return self.func.__doc__

            def __get__(self, obj, objtype):
               '''Support instance methods.'''
               return functools.partial(self.__call__, obj)

         @memoized
         def fibonaccim(n):
            "Return the nth fibonacci number."
            print('in fib', n)
            if n in (0, 1):
               return n
            return fibonaccim(n-1) + fibonaccim(n-2)

         # now no redundant calls
         fibonaccim(8)
```

```
in fib 8
in fib 7
in fib 6
in fib 5
in fib 4
in fib 3
in fib 2
in fib 1
in fib 0
```

```
Out[11]: 21
```

```
In [12]: # functools has a better memo decorator

         import functools

         # maxsize=an int will limit the size of the cache

         @functools.lru_cache(maxsize=None)
         def fiblru(n):
            "Return the nth fibonacci number."
            print('in fib', n)
            if n in (0, 1):
               return n
            return fiblru(n-1) + fiblru(n-2)

         fiblru(8)
```

```
in fib 8
in fib 7
in fib 6
in fib 5
in fib 4
in fib 3
in fib 2
in fib 1
in fib 0
```

```
Out[12]: 21
```

```
In [13]: fiblru(8)
```

```
Out[13]: 21
```

```
In [14]: # info about the cache

         fiblru.cache_info()
```

```
Out[14]: CacheInfo(hits=7, misses=9, maxsize=None, currsize=9)
```

```
In [15]: # can clear the cache

         fiblru.cache_clear()
```

```
In [6]: fiblru.cache_info()
```

```
Out[6]: CacheInfo(hits=0, misses=0, maxsize=None, currsize=0)
```

5 Standard Library of Decorators

- some useful things