Decorator Pattern

- In object-oriented programming, the decorator pattern is a design pattern that allows new/additional behaviour to be added to an existing class dynamically.
- In Python one cannot say that to be the same with the Decorator; even though one can achieve the same functionality with decorators in python.
- So, what are decorators IN Python?

Functions taking Functions

- Functions can have pointers to other functions as parameters.
- A function which can take another function as its parameter and can achieve something there by could be mainly classified as a decorator.
  See example.

```
>>> def ourdecorator2(foo):
...     def newfoo():
...         print "We are to call", foo.__name__
...         return foo()
...     return newfoo
...
>>>
>>> foo = ourdecorator2(foo1)
>>>
>>> foo()
We are to call foo1
 Hello World
>>>
```

```
>>>
>>> def ourdecorator(foo):
...         print "We are to call", foo.__name__
...         return foo()
...
>>> def fool():
...         print "Hello World"
...
>>>
>>> foo  = ourdecorator(fool)
We are to call fool
Hello World
>>>
>>>
```

Similar to Macros

- Decorators are similar to MACROS in other programming languages
- They are usually used to make a wrapper around functions
- And of course, classes too.

Usage, then Creation

- Function decorators are placed above the function with the key-character '@'

```
@thedecorator
def foo():
....
```

- The interpreter compiles `foo` and calls the decorator with that as argument.
- The result of that replaces the code for `foo`

How to implement decorator

- Could be Functions or Classes.
- The condition is that whatever the decorator returns, that should be callable.
- An object is callable, if the method `__call__` is implemented.

```
class theDecorator(object):
    def __init__(self, f):
        print "inside theDecorator.__init__()"
        f()
    def __call__(self):
        print "inside theDecorator.__call__()"


@theDecorator
def foobar():
    print "inside foobar()"

print "Finished decorating foobar()"

foobar()
```

```
>>> import decorators.py
inside theDecorator.__init__()
```

```
3 inside foobar()
4 Finished decorating foobar()
5 inside theDecorator.__call__()
```

Observation

- From the output, it is clear that the `init` is called when the decorator is used.
- So, usually, the call to the function is done only in the `call` function.
- Once a function is decorated, the behaviour totally changes. The call goes only to the decorated code. (line number 4 of the output)

```
1  class loggerdeco(object):
2
3      def __init__(self, f):
4          self.f = f
5
6      def __call__(self):
7          print "Entering", self.f.__name__
8          self.f()
9          print "Exited", self.f.__name__
10
11 @loggerdeco
12 def func1():
13     print "inside func1()"
14
15 @loggerdeco
16 def func2():
17     print "inside func2()"
```

```
1
2 func1()
3 func2()
4 Entering func1
5
6
7 inside func1()
8 Exited func1
9 Entering func2
10 inside func2()
11 Exited func2
```

Using Functions

- The same can be achieved using functions, instead of classes.
- The decorator functions usually enclose the decorated function in between the decoration.
- This is done inside a subfunction (equivalent of `__call__` and the pointer to the subfunction is returned.

```
1  def loggerdeco(f):
2      def new_f():
3          print "Entering", f.__name__
4          f()
5          print "Exited", f.__name__
6      return new_f
7
8  @loggerdeco
9  def func1():
10     print "inside func1()"
11
12 @loggerdeco
13 def func2():
14     print "inside func2()"
```

```
1  func1()
2  func2()
3  print func1.__name__
4
5
6  Entering func1
7  inside func1()
8  Exited func1
9  Entering func2
10 inside func2()
11 Exited func2
12 new_f
```

Observation

- The name of the functions have been changed to `new_f`.
- This can be changed by reassigning `new_f.__name__ = f.__name__`

There are many cool uses of decorators. You can see more examples at `http://wiki.python.org/moin/PythonDecorat`

```
1  class memoized(object):
2    def __init__(self, func):
3      self.func = func
4      self.cache = {}
5    def __call__(self, *args):
6      try:
7        return self.cache(args)
8      except KeyError:
9        self.cache(args) = value = self.func(*args)
10       return value
11     except TypeError:
12       return self.func(*args)
13   def __repr__(self):
14     return self.func.__doc__
15
16 @memoized
17 def fibonacci(n):
18   "Return the nth fibonacci number."
```

```
19     if n in (0, 1):
20         return n
21     return fibonacci(n-1) + fibonacci(n-2)
22
23 for i in xrange(1, 100, 9):
24     print fibonacci(i)
```

```
(sadanand@lxmayr10 @ ~)time python memorized.py
1
55
4181
317811
24157817
1836311903
139583862445
10610209857723
806515533049393
6130579072161159 1
466004661037553030 9

real    0m0.014s
user    0m0.008s
sys     0m0.000s
(sadanand@lxmayr10 @ ~)
```

Descriptors

- Any object which defines the methods __get__(), __set__(), or __delete__().
- Normally, using a.b to get, set or delete an attribute looks up the object named b in the class dictionary for a, but if b is a descriptor, the respective descriptor method gets called.

```
1  class ourDescriptor(object):
2
3      def __init__(self, initval=None, name='var'):
4          self.val = initval
5          self.name = name
6
7      def __get__(self, obj, objtype):
8          print 'Retrieving', self.name
9          return self.val
10
11     def __set__(self, obj, val):
12         print 'Updating' , self.name
13         self.val = val
```

```
>>> class MyClass(object):
        x = ourDescriptor(10, 'var "x"')
        y = 5

>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

property((fget(, fset(, fdel(, doc)))))

fget is a function for getting the attribute value,

fset is a function for setting

fdel a function for deling, the attribute.

doc is the doc string

Property can also be used as a decorator. `@property, @x.setter, @deleter for the get/set/del function for the attribute x`

```
class C(object):
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm 'x'")
```

Static Variables and Methods

- A static variable in a class has always the same value, independent of the instances.
- Static variables are class variables, they belong to the class than to the instances
- They are accessed by the name of the Class, rather than the instance.

```
1  class myStatic:
2      instances = 0
3      def __init__(self):
4          myStatic.instances += 1
5
6      def howmany(self):
7          return myStatic.instances
8
9  x = myStatic()
10 print x.howmany()
11 y = myStatic()
12 print y.howmany()
13 print x.howmany()
14 ----------------
15 print 1, 2, 2
```

Static Methods

- They have the same return value independent of the class instance
- They too belong to the class as much as the variables
- They don't have the `self` parameter
- For the same reason, they cannot access any of the `self.*` objects.
- The keyword is a decorator named `@staticmethod`

```
1  class myStatic:
2      instances = 0
3
4      def __init__(self):
5          myStatic.instances += 1
6
7      @staticmethod
8      def howmany():
9          return myStatic.instances
10
11
12 x = myStatic()
13 print myStatic.howmany()
14 y = myStatic()
15 print myStatic.howmany()
16 ==========================
17 prints 1, 2 as expected.
```

Classmethod

- `@classmethod` is perhaps a special thing for python.
- The methods decorated with this gets as the initial variable a class which is the original class (not the instance)
- That helps the function to act like a normal method of the class, by accepting all the attributes and treat them as static as well.

```
1  class myStatic:
2      instances = 0
3      def __init__(self):
4          self.addinstance()
5
6      @classmethod
7      def howmany(cls):
8          return cls.instances
9      @classmethod
10     def addinstance(cls):
11         cls.instances += 1
12
13 x = myStatic()
14 print myStatic.howmany()
15 y = myStatic()
16 print myStatic.howmany()
17 ================================
18 prints 1, 2 as expected.
```

```
1  class myStatic:
2      instances = 0
3
4      def __init__(self):
5          self.addinstance()
6
7      @classmethod
8      def howmany(cls):
9          return cls.instances
10
11     @classmethod
12     def addinstance(cls):
13         cls.instances += 1
14
15     def nastything(self):
16         print "trying to be nasty"
17         self.instances = -1
```

```
1  x = myStatic()
2  x.nastything()
3  print myStatic.howmany()
4  print x.howmany()
5  y = myStatic()
6  x.nastything()
7  print myStatic.howmany()
8  ======================
9  trying to be nasty
10 1
11 1
12 trying to be nasty
13 2
```

Single Instance

- The static methods can be used to create a singleton object/pattern
- They are classes for which there is only one instance at any given time.
- They could be implemented using
  1. The class instance could lie in a static variable
  2. The method which gets the instance can be made static.

Anonymous Classes

- New classes could be defined inside functions and returned.
- Such are called anonymous classes
- Anonymous classes can also be created using `classobj`

```
1  def createclass(name):
2      class myClass:
3          def __init__ (self):
4              self.name = name
5          def whoareyou(self):
6              print self.name
7
8      return myClass
9
10 Creator = createclass('iAmCreator')
11 first = Creator()
12 first.whoareyou()
13 ========================
14 prints iAmCreator as expected.
```

```
1  from new import classobj
2  class foo:
3      def __init__ (self):
4          self.x = 'x'
5
6  foo2 = classobj('foo2', (foo,),
7    {'bar': lambda self, x: 'got ' + str(x)})
8
9  print foo2().bar(3)
10 print foo2().x
11 ================
12 prints got2, x as expected.
```

Meta Classe

- Not in the scope of our course.
- Creating tailormade classes / customized ones.
  `__metaclass__`

Some Philosophy?

If you'd like to know some python philosophy, then you may import the module `this`

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Tab Complete

- Getting Tab Complete (like Bash) in python prompt.

```
1
2 try:
3     import readline
4 except ImportError:
5     print "Unable to load readline module."
6 else:
7     import rlcompleter
8     readline.parse_and_bind("tab: complete")
```