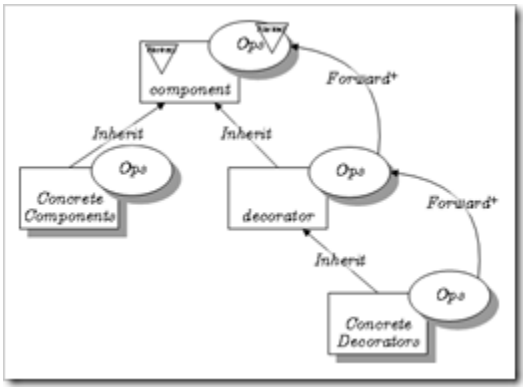
Aspect Oriented Programming in Python using Decorators

It always amazes me how some tasks are easier using dynamic languages. AOP (Aspect Oriented Programming) is just one more thing that can be done easily using Python – at least to some extent.

Python has a powerful language feature that makes it all happen – called "Function Decorator" and this is how you use it:

The Decorator pattern



If I'm forced to "name names" the design pattern used is called "Decorator" and is used to extend a class using composition at runtime.

In simple term I create a class and use it to wrap another class and by doing so I add functionality to it.

So what do it have to do with AOP? simple if I want to add an Aspect to a method I can do it by "decorating" that method with another method. The following code catch and "log" exception from methods:

```
def decorator(function):
   def inner():
      try:
         return function()
      except:
         print("Exception caught")

   return inner

def someMethod():
   print "someFunction called - going to throw exception"
   raise Exception()

# Run method
decoratedMethod = decorator(someMethod)
decoratedMethod()
```

It works but something is missing. wrapping a method seems a bit awkward, it's a good thing that Python has a more elegant solution – decorators.

Enter Python Decorators

Using Decorators is easy – simply add @ with the method/class before the "decorated" method:

```
def decorator(function):
   def inner():
      try:
         return function()
      except:
         print("Exception caught")

   return inner

@decorator
def someMethod():
   print "someFunction called - going to throw exception"
   raise Exception()

#Run method - look Ma no hands
someMethod()
```