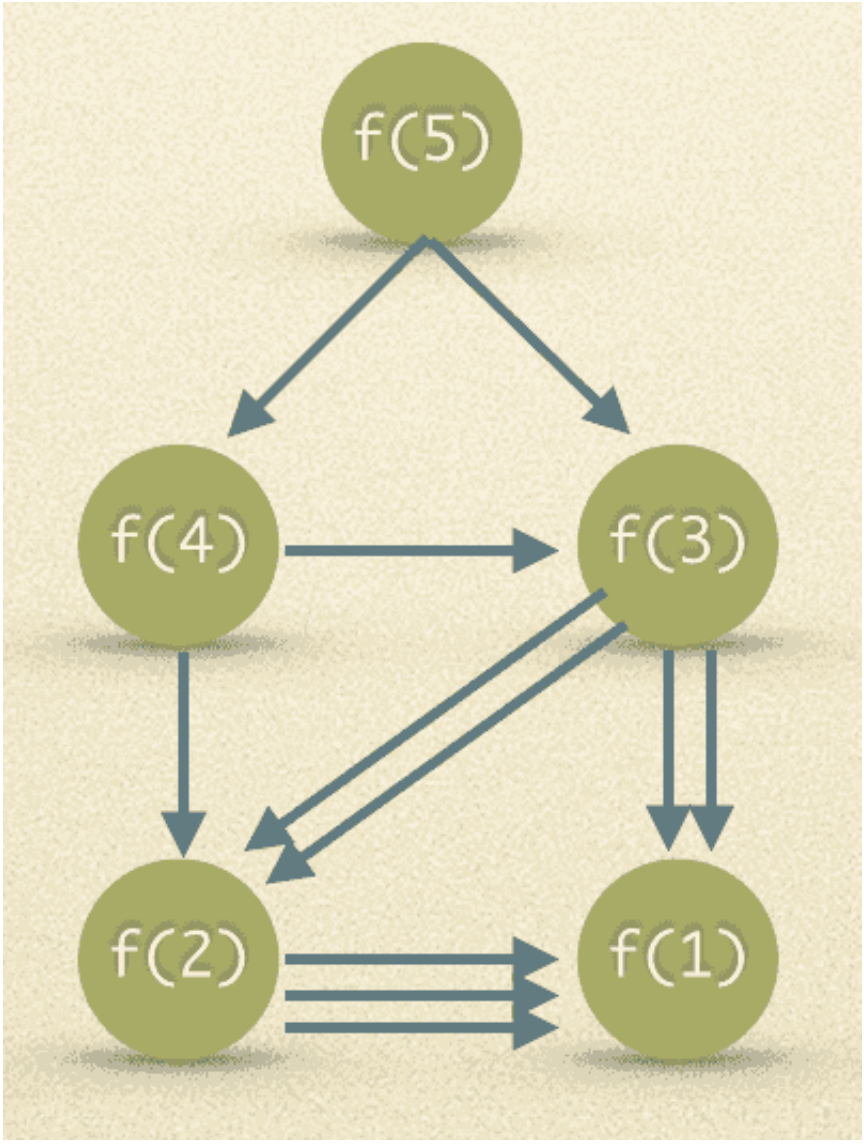
python装饰器详解

之前用python简单写了一下斐波那契数列的递归实现（如下），发现运行速度很慢。

```
def fib_direct(n):
    assert n > 0, 'invalid n'
    if n < 3:
        return n
    else:
        return fib_direct(n - 1) + fib_direct(n - 2)
```

然后大致分析了一下fib_direct(5)的递归调用过程，如下图：



递归调用

可以看到多次重复调用，因此效率十分低。进一步，可以算出递归算法的时间复杂度。T(n) = T(n-1) + T(n-2)，用常系数线性齐次递推方程的解法，解出递推方程的特征根，特征根里最大的n次方就是它的时间复杂度O(1.618^n)，指数级增长。

为了避免重复调用，可以适当地做缓存，python的装饰器可以完美的完成这一任务。

装饰器：基础

python中一切都是对象，这里需要强调函数是对象。为了更好地理解函数也是对象，下面结合代码片段来说明这一点。

```
1
2
3
4
5
6  def shout(word="yes"):
7      return word.capitalize() + "!"
8  print shout()
9  """
10 As an object, you can assign the function to a variable like any other object.
11 Notice we don't use parentheses: we are not calling the function,
12 we are putting the function "shout" into the variable "scream".
13 """
14 scream = shout
15 print scream()
16 """
17 More than that, it means you can remove the old name 'shout',
18 and the function will still be accessible from 'scream'.
19 """
20 del shout
21 try:
22     print shout()
23 except NameError, e:
24     print e
25 print scream()
26
27
28
29
30
```

因为函数是对象，所以python中函数还有一个有趣的特性：函数可以被定义在另一个函数中。下面来看一个简单的例子。

```
1
2
```

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

```
def talk():
    def whisper(word="yes"):
        return word.lower() + "..."
    print whisper()
"""
You call "talk", that defines "whisper" EVERY TIME you call it,
then "whisper" is called in "talk".
"""
talk()
try:
    print whisper()
except NameError, e:
    print e
```

函数引用

前面已经知道函数是对象。那么：

1. 可以被赋给另一个变量
2. 可以被定义在另一个函数里

这也意味着，一个函数可以返回另一个函数，下面看一个简单的例子。

```
1
2
3
4
5
6
7
8
```

```
def get_talk(kind="shout"):
    def whisper(word="yes"):
```

```
        return word.lower() + "..."
    def shout(word="yes"):
        return word.capitalize() + "!"
    return whisper if kind == "whisper" else shout
talk = get_talk()
print talk
print talk()
print get_talk("whisper")()
```

我们来进一步挖掘一下函数的特性，既然可以返回函数，那么我们也可以把函数作为参数传递。

```

def whisper(word="yes"):
    return word.lower() + "..."
def do_something_before(func):
    print "I do something before."
    print "Now the function you gave me:\n", func()
do_something_before(whisper)
"""outputs
I do something before.
Now the function you gave me:
yes...
"""
```

现在，了解装饰器所需要的所有要点我们已经掌握了，通过上面的例子，我

们还可以看出，装饰器其实就是封装器，可以让我们在不修改原函数的基础上，在执行原函数的前后执行别的代码。

手工装饰器

下面我们手工实现一个简单的装饰器。

```
def my_shiny_new_decorator(a_function_to_decorate):
    """
    Inside, the decorator defines a function on the fly: the wrapper.
    This function is going to be wrapped around the original function
    so it can execute code before and after it.
    """
    def the_wrapper_around_the_original_function():
        """
        Put here the code you want to be executed BEFORE the original
        function is called
        """
        print "Before the function runs"
        a_function_to_decorate()
        """
        Put here the code you want to be executed AFTER the original
        function is called
        """
        print "After the function runs"
    """
    At this point, "a_function_to_decorate" HAS NEVER BEEN EXECUTED.
    We return the wrapper function we have just created.
    The wrapper contains the function and the code to execute before
    and after. It’s ready to use!
    """
    return the_wrapper_around_the_original_function
```

```
def a_stand_alone_function():
    print "I am a stand alone function, don't you dare modify me"
a_stand_alone_function()
"""
Well, you can decorate it to extend its behavior.
Just pass it to the decorator, it will wrap it dynamically in
any code you want and return you a new function ready to be used:
"""
a_stand_alone_function_decorated = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function_decorated()
"""outputs:
Before the function runs
I am a stand alone function, don't you dare modify me
After the function runs
"""
```

现在，如果我们想每次调用a_stand_alone_function的时候，实际上调用的是封装后的函数a_stand_alone_function_decorated，那么只需要用 a_stand_alone_function 去覆盖 my_shiny_new_decorator返回的函数即可。也就是：

```
a_stand_alone_function = my_shiny_new_decorator(a_stand_alone_function)
```

装饰器阐述

对于前面的例子，如果用装饰器语法，可以添加如下：

```
```

```
@my_shiny_new_decorator
def another_stand_alone_function():
    print "Leave me alone"
another_stand_alone_function()
"""outputs:
Before the function runs
Leave me alone
After the function runs
"""
```

对了，这就是装饰器语法，这里的`@my_shiny_new_decorator`是`another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)`的简写。

装饰器只是装饰器设计模式的python实现，python还存在其他几个经典的设计模式，以方便开发，例如迭代器iterators。

当然了，我们也可以嵌套装饰器。

```


def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "<\______/>"
    return wrapper
def ingredients(func):
    def wrapper():
        print "#tomatoes#"
        func()
        print "~salad~"
    return wrapper
def sandwich(food="--ham--"):
```

```
    print food
sandwich()
sandwich = bread(ingredients(sandwich))
sandwich()
"""outputs:
</''''''\>
 #tomatoes#
 --ham--
 ~salad~
<\______/>
"""
```

用python的装饰器语法，如下：

```
@bread
@ingredients
def sandwich_2(food="--ham_2--"):
    print food
sandwich_2()
```

放置装饰器的位置很关键。

```
@ingredients
@bread
def strange_sandwich(food="--ham--"):
    print food
strange_sandwich()
"""outputs:
#tomatoes#
</''''''\>
```

```
 --ham--
<\______/>
 ~salad~
"""
```

装饰器高级用法

给装饰器函数传递参数

当我们调用装饰器返回的函数时，其实是在调用封装函数，给封装函数传递参数也就同样的给被装饰函数传递了参数。

```

def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print "I got args! Look:", arg1, arg2
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
"""
Since when you are calling the function returned by the decorator, you are
calling the wrapper, passing arguments to the wrapper will let it pass them to
the decorated function
"""
@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print "My name is", first_name, last_name
print_full_name("Peter", "Venkman")
"""outputs:
I got args! Look: Peter Venkman
My name is Peter Venkman
"""

```

装饰方法

python中函数和方法几乎一样，除了方法中第一个参数是指向当前对象的引用(self)。这意味着我们可以为方法创建装饰器，只是要记得考虑self。

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3
        return method_to_decorate(self, lie)
    return wrapper
class Lucy(object):
    def __init__(self):
        self.age = 32
    @method_friendly_decorator
    def sayYourAge(self, lie):
        print "I am %s, what did you think?" % (self.age + lie)
l = Lucy()
l.sayYourAge(-3)
```

我们还可以创建一个通用的装饰器，可以用于所有的方法或者函数，而且不用考虑它的参数情况。这时候，我们要用到*args, **kwargs。

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print "Do I have args?:"
        print args
```

```
        print kwargs
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments
```

另外还有一些高级用法，这里不做详细说明，可以在How can I make a chain of function decorators in Python? 进一步深入了解装饰器。

functools.wraps

装饰器封装了函数，这使得调试函数变得困难。不过在python 2.5引入了`functools`模块，它包含了`functools.wraps()`函数，这个函数可以将被封装函数的名称、模块、文档拷贝给封装函数。有趣的是，functools.wraps是一个装饰器。为了更好地理解，看以下代码：

```
def foo():
    print "foo"
print foo.__name__
def bar(func):
    def wrapper():
        print "bar"
        return func()
    return wrapper
@bar
def foo():
    print "foo"
print foo.__name__
import functools
```

```
def bar(func):
    @functools.wraps(func)
    def wrapper():
        print "bar"
        return func()
    return wrapper
@bar
def foo():
    print "foo"
print foo.__name__
```

为何装饰器那么有用

让我们回到本篇文章开始的问题上，重复调用导致递归的效率低下，因此考虑使用缓存机制，空间换时间。这里，就可以使用装饰器做缓存，看下面代码:

```


from functools import wraps
def cache(func):
    caches = {}
    @wraps(func)
    def wrap(*args):
        if args not in caches:
            caches[args] = func(*args)
        return caches[args]
    return wrap
@cache
def fib_cache(n):
```

```
14      assert n > 0, 'invalid n'
15      if n < 3:
16          return 1
17      else:
18          return fib_cache(n - 1) + fib_cache(n - 2)
19
20
```

这样递归中就不会重复调用，效率也会提高很多。具体可以看这里，从执行时间很容易看出做了缓存之后速度有了很大的提升。装饰器还可以用来扩展外部接口函数(通常你不能修改它)的功能，或者用来调试函数。其实，装饰器可以用于各种各样的场合！

python本身提供了一些装饰器：property,staticmethod，等等。另外，Django使用装饰器去管理缓存和权限。

更多阅读

计算斐波纳契数，分析算法复杂度
How can I make a chain of function decorators in Python?
Python装饰器与面向切面编程
how to use args and kwargs in python?
Fibonacci, recursion and decorators