Decorators

Basic decorator syntax

Python *decorators* allow us to modify function and class definitions with a special syntax.

In [3]:
```
def log_function_call(function):
    def wrapper(*args, **kwargs):
        print 'Calling %s(*%r, **%r)' % (function, args, kwargs)
        return function(*args, **kwargs)
    print 'returning wrapped %s' % function
    return wrapper

def myfunction(a, b):
    print 'myfunction(%r, %r)' % (a,b)

myfunction = log_function_call(myfunction)
```

```
returning wrapped <function myfunction at 0x27eaf50>
```

In [4]:
```
myfunction('avalue', 'bvalue')
```

```
Calling <function myfunction at 0x27eaf50>(*('avalue', 'bvalue'), **{})
myfunction('avalue', 'bvalue')
```

A nicer syntax for this uses the `@` sign:

In [6]:
```
@log_function_call
def myfunction(a,b):
    print 'myfunction(%r, %r)' % (a,b)

myfunction('avalue', 'bvalue')
```

```
returning wrapped <function myfunction at 0x27eaed8>
Calling <function myfunction at 0x27eaed8>(*('avalue', 'bvalue'), **{})
myfunction('avalue', 'bvalue')
```

We can also decorate class definitions:

In [8]:
```
def add_myproperty(cls):
    cls.myproperty = 'Magically added by decorator'
    return cls

@add_myproperty
class MyClass(object):
    def __init__(self, a, b):
        self._a = a
        self._b = b
    def __repr__(self):
        return 'MyClass(%r, %r)' % (a,b)

MyClass.myproperty
```

Out[8]:
```
'Magically added by decorator'
```

Useful decorators

In [10]:
```
class MyClass(object):
    @property
    def myproperty(self):
        print 'Calling myproperty'
        return 'myvalue'

x = MyClass()
print x.myproperty
```

```
Calling myproperty
myvalue
```

In [13]:
```
class MyClass(object):

    def __init__(self):
        self._value = None

    @property
    def myproperty(self):
        print 'Getting myproperty'
        return self._value

    @myproperty.setter
    def myproperty(self, value):
        print 'Setting myproperty'
        self._value = value


x = MyClass()
print x.myproperty
print

x.myproperty = 5
print x.myproperty
```

```
Getting myproperty
None

Setting myproperty
Getting myproperty
5
```

In [14]:
```
class MyClass(object):

    def do_something_with_instance(self):
        print 'Instance method on', self

    @classmethod
    def do_something_with_class(cls):
        print 'Class method on', cls

    @staticmethod
    def do_something_without_either():
        print 'Static method'

x = MyClass()
x.do_something_with_instance()
```

```
Instance method on <__main__.MyClass object at 0x27f3dd0>
```

In [15]:
```
x.do_something_with_class()
```

```
Class method on <class '__main__.MyClass'>
```

In [16]:
```
MyClass.do_something_with_class()
```

```
Class method on <class '__main__.MyClass'>
```

In [17]:
```
x.do_something_without_either()
```

```
Static method
```

In [18]:
```
MyClass.do_something_without_either()
```

```
Static method
```

Building your own decorators

In [25]:
```
def log_function_call(function):
    def wrapper(*args, **kwargs):
        print 'Calling %s(*%r, **%r)' % (function, args, kwargs)
        return function(*args, **kwargs)
    print 'returning wrapped %s' % function
    return wrapper

@log_function_call
def myfunction(a, b):
    print 'myfunction(%r, %r)' % (a,b)

myfunction(1,2)
```

```
returning wrapped <function myfunction at 0x27f4d70>
Calling <function myfunction at 0x27f4d70>(*(1, 2), **{})
myfunction(1, 2)
```

In [27]:
```
def log_function_call(message):
    def decorator(function):
        def wrapper(*args, **kwargs):
            print '%s: %s(*%r, **%r)' % (message, function, args, kwargs)
            return function(*args, **kwargs)
        print 'returning wrapped %s' % function
        return wrapper
    print 'returning decorator(%r)' % message
    return decorator

@log_function_call('log1')
def myfunction(a, b):
    print 'myfunction(%r, %r)' % (a,b)

myfunction(1,2)
```

```
returning decorator('log1')
returning wrapped <function myfunction at 0x27f4d70>
log1: <function myfunction at 0x27f4d70>(*(1, 2), **{})
myfunction(1, 2)
```

To simplify things a bit, we can also use the magic `__call__` method to define a decorator that takes arguments:

In [28]:
```
class log_function_call(object):
    def __init__(self, message):
        self._message = message
    def __call__(self, function):
        def wrapper(*args, **kwargs):
            print '%s: %s(*%r, **%r)' % (
                self._message, function, args, kwargs)
            return function(*args, **kwargs)
        return wrapper

@log_function_call('log1')
def myfunction(a, b):
    print 'myfunction(%r, %r)' % (a,b)

myfunction(1,2)
```

```
log1: <function myfunction at 0x27eaed8>(*(1, 2), **{})
myfunction(1, 2)
```

One useful decorator to build is one that *memoizes* function results:

In [38]:
```
def memoize(function):
    cache = {}
    def wrapper(*args, **kwargs):
        cache_key = (args, tuple(sorted(kwargs.items())))
        if cache_key in cache:
            print '-- return cached value for', cache_key
            return cache[cache_key]
        result = function(*args, **kwargs)
        cache[cache_key] = result
        return result
    return wrapper

@memoize
def my_function(a, b):
    print 'Calling my_function(%r,%r)' % (a,b)

my_function(1,2)
my_function(1,2)
my_function(1,2)
my_function(3,4)
my_function(5,6)
```

```
Calling my_function(1,2)
-- return cached value for ((1, 2), ())
-- return cached value for ((1, 2), ())
Calling my_function(3,4)
Calling my_function(5,6)
```

Exercises

- Write a class that uses `@property` to provide read-only access to an underlying "private" attribute
- Write a decorator that takes a logger and logs all entries/exits of a function
- Write a decorator that opens a file at the beginning of a function and closes it at the end, passing the opened file as the first argument of the inner function.