Python Decorators

Arguments

Variable arguments and variable keyword arguments are necessary for functions that accept arbitrary parameters.

*args and **kw

```
>>> def param_func(a, b='b', *args, **kw):
...     print [x for x in [a, b, args, kw]]
>>> param_func(2, 'c', 'd', 'e,')
[2, 'c', ('d', 'e,'), {}]
>>> args = ('f', 'g')
>>> param_func(3, args)
[3, ('f', 'g'), (), {}]
>>> param_func(4, *args) # tricksey!
[4, 'f', ('g',), {}]
>>> param_func(5, 'x', *args)
[5, 'x', ('f', 'g'), {}]
>>> param_func(6, **{'foo':'bar'})
[6, 'b', (), {'foo': 'bar'}]
```

The splat operator (* in a function *invocation*) is the same as enumerating the values in the sequence.

The following are equivalent:

```
>>> param_func(*args) # tricksey!
['f', 'g', (), {}]
```

and:

```
>>> param_func(args[0], args[1])
['f', 'g', (), {}]
```

Closures

Closures are useful as function generators:

```
>>> def add_x(x):
...     def adder(y):
...         return x + y
...     return adder

>>> add_5 = add_x(5)
>>> add_7 = add_x(7)
>>> add_5(10)
15
>>> add_7(10)
17
```

Closures are also useful for decorators

Decorator Template

```
>>> import functools

>>> def decorator(func_to_decorate):
...     @functools.wraps(func_to_decorate)
...     def wrapper(*args, **kw):
...         print "before invocation"
...         result = func_to_decorate(*args, **kw)
...         print "after invocation"
...         return result
...     return wrapper
```

Syntactic Sugar

The following are the same:

```
>>> @decorator
... def foo():
...     print "hello"
```

and:

```
>>> def foo():
...     print "hello"
>>> foo = decorator(foo)
```

Invoking a decorated function:

```
>>> foo()
before invocation
hello
after invocation
```

Parameterized decorators (need 2 closures)

```
>>> def limit(length):
...     def decorator(function):
...         @functools.wraps(function)
...         def wrapper(*args, **kw):
...             result = function(*args, **kw)
...             result = result[:length]
...             return result
...         return wrapper
...     return decorator
```

The following are the same:

```
>>> @limit(5) # notice parens
... def echo(foo):
...    return foo
```

and:

```
>>> def echo(foo):
...     return foo
>>> echo = limit(5)(echo)

>>> echo('123456')
'12345'
```

Functions can decorate with `@limit(1)` or `@limit(20)`...

Class instances as decorators

```
>>> class Decorator(object):
...     # in __init__ set up state
...     def __call__(self, function):
...         @functools.wraps(function)
...         def wrapper(*args, **kw):
...             print "before func"
...             result = function(*args, **kw)
...             print "after func"
...             return result
...         return wrapper

>>> decorator2 = Decorator()
>>> @decorator2
... def nothing(): pass
```

Decorating classes

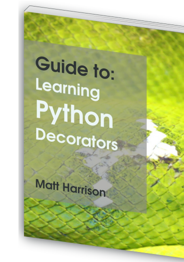
```
>>> class Cat(object):
...     def __init__(self, name):
...         self.name = name
...
...     @decorator
...     def talk(self, txt):
...         print '{0} said, "{1}"'.format(
...             self.name, txt)
...
...     @decorator2
...     def growl(self, txt):
...         print txt.upper()

>>> cat = Cat('Fred')
>>> cat.talk("Meow.")
before invocation
Fred said, "Meow."
after invocation

>>> cat.growl("GRRR.")
before func
GRRR.
after func
```

More Details

For an in depth explanation of the above check out my ebook, *Guide to: Learning Python Decorators*. http://hairysun.com/books/decorators/



© *Matt Harrison* 2012
http://hairysun.com/
Note that examples are illustrated as if they were done in a terminal.