Decorators

- Do something before, during, and/or after some code
- Goal: reduce boilerplate code
- Extend the behavior of a function without modifying it
- “Design pattern that allows behavior to be added to an existing object dynamically.”

Functions: review

```
def foo():
    """Docstring"""
    print 'Hello!'
```

```
>>> foo
>>> foo()
```

```
>>> bar = foo
>>> bar.__name__
'foo'
```

Parameters: positional, keyword, variable (*args), variable keyword (**kwargs)

```
def get_foo():
    return foo
```

```
>>> dir(foo)
```

```
def adder():
    def add(x,y):
        return x + y
    return add

>>> adder()
>>> adder()(2,4)
```

Generic decorator pattern

```
def mydecorator(some_function):
    def inner_function(*args, **kw):
        # do some stuff before
        result = some_function(*args,
                               *kw)
        # do some stuff after
        return result
    return inner_function
```

To use:

```
@mydecorator
def some_function():
    ...
```

Equivalent syntax

```
@mydecorator
def myfunc():
    pass
```

```
def myfunc():
    pass
myfunc = mydecorator(myfunc)
```

```python
def verbose(my_func):
    def inner_function(*args, **kwargs):
        print "before", my_func.__name__
        result = my_func(*args, **kwargs)
        print "after", my_func.__name__
        return result
    return inner_function
```

```python
@verbose
def print_message():
    print "Hello there!"
```

```
>>> print_message()
before print_message
Hello there!
after print_message
```

Decorators

All credit goes to Matt Harrison for this example!

Flask example

```python
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

When to use decorators

- Minimize boilerplate code and simplify functions
- Logging
- Error handling
- Caching expensive calculations
- Retrying functions that might fail