Decorators

Functions That Make Functions

CSCI 101

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

In this case, the first style is preferred. It's a bit easier to read, not to mention it's actually named.

*args, **kwargs

Python allows you to define functions that take a variable number of positional (*args) or keyword (**kwargs) arguments. In principle, this really just works like tuple expansion/collection.

*args, **kwargs

Python allows you to define functions that take a variable number of positional (*args) or keyword (**kwargs) arguments. In principle, this really just works like tuple expansion/collection.

```python
def crazyprinter(*args, **kwargs):
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print("{}={}".format(k, v))

crazyprinter("hello", "cheese", bar="foo")
# hello
# cheese
# bar=foo
```

Decorators

`@property` as we just saw is what is called a decorator. Decorators are really just a pretty way to wrap functions using functions that return functions.

Decorators

`@property` as we just saw is what is called a decorator. Decorators are really just a pretty way to wrap functions using functions that return functions.

Both the following are equivalent:

```
@logging
def foo(bar, baz):
    return bar + baz - 42

# equivalent to...
def foo(bar, baz):
    return bar + baz - 42
foo = logging(foo)
```

Defining Decorators

When defining wrapper functions, you should decorate it with `wraps` from `functools`, this will keep attributes about the function.

```python
from functools import wraps

def logging(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(result)
        return result
    return wrapper
```

Decorators in the Wild: Dynamic Programming

`lru_cache` from `functools` can be a quick way to make a recursive function with a recurrence relation fast. Here's an example:

Decorators in the Wild: Dynamic Programming

lru_cache from functools can be a quick way to make a recursive function with a recurrence relation fast. Here's an example:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Decorators in the Wild: Welford's Equations

Welford's Equations are a one-pass mean and standard deviation algorithm. One important property is that we won't have to store the results in a list.

Decorators in the Wild: Welford's Equations

Welford's Equations are a one-pass mean and standard deviation algorithm. One important property is that we won't have to store the results in a list.

Our goal will be to implement a decorator we can use like this:

```python
@Welford
def diceroll(u):
    return int(u * 6) + 1

# call diceroll with some u's in (0, 1)

print(diceroll.mean, diceroll.stdev)
```

Decorators in the Wild: Implementing Welford

The key here is that we can make callable objects using `__call__`.

```
from functools import update_wrapper
from math import sqrt

class Welford:
    def __init__(self, f):
        self.f = f
        update_wrapper(self, f)
        self.mean = 0
        self.v = 0
        self.trials = 0

    def __call__(self, *args, **kwargs):
        r = self.f(*args, **kwargs)
        self.trials += 1
        d = r - self.mean
        self.v += d**2 * (self.trials - 1)/self.trials
        self.mean += d/self.trials
        return r

    @property
    def stdev(self):
        return sqrt(self.v/self.trials) if self.trials else 0
```

More Decorator Tricks

- Decorators can wrap classes as well as functions. A practical example might be creating a decorator which adds attributes of a class to a database (a `@model` decorator?)

More Decorator Tricks

- Decorators can wrap classes as well as functions. A practical example might be creating a decorator which adds attributes of a class to a database (a `@model` decorator?)
- When multiple decorators are typed, they are applied bottom-up.