Decorators in Python

Need to measure timing of algorithms

- Measure execution time
  - Wall-clock timing:
    - Import a clock or time module
    - Save current time
    - Execute function
    - Save current time
    - Difference between saved times is the duration

Need to measure timing of algorithms

- We measure the *implementation* of an algorithm
  - Wall-clock times are inaccurate:
    - System is doing other things
    - Measuring introduces additional overhead

Need to measure timing of algorithms

```
import time

for i in range(1,25):
    print(i)
    for j in range(20):
        start_time = time.perf_counter()
        for _ in range(50):
            x = fibonacci(i)
        duration = (time.perf_counter() - start_time)/50
        print("{:12.10f}".format(duration))
    print("\n")
```

Decorators

- Python uses decorators to allow changing functions
- A decorator is implemented by:
  - Creating a function of a function that returns the amended function

Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
    arg_string = ', '.join(repr(arg) for arg in args)
    print('Function {} with arguments {} ran
           in {} seconds'.format(
        name, arg_string, duration))
    return result
return clocked
```

Decorators

- Decorator takes a function with positional arguments as function
- Decorator defines a new version of the argument function
- And returns it.

Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
    arg_string = ', '.join(repr(arg) for arg in args)
    print('Function {} with arguments {} ran
           in {} seconds'.format(
        name, arg_string, duration))
    return result
return clocked
```

Decorators

- To use a decorator, just put its name on top of the function definition
  - Decorator generator is executed when module is imported (or generator is defined)
  - When decorated function is defined, the modified version is created

Decorators

```
@timeit
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Decorators

- If we execute this function, we get to see how often fibonacci is called on arguments already executed

```
>>> fibonacci(10)
Function fibonacci with arguments 1 ran in 5.140000070014139e-07 seconds
Function fibonacci with arguments 0 ran in 1.0870000011209413e-06 seconds
Function fibonacci with arguments 2 ran in 0.1692790839999958 seconds
Function fibonacci with arguments 1 ran in 1.2330000060956081e-06 seconds
Function fibonacci with arguments 3 ran in 0.2676633440000131 seconds
Function fibonacci with arguments 1 ran in 9.8000001003129e-07 seconds
Function fibonacci with arguments 0 ran in 1.0470000120221812e-06 seconds
Function fibonacci with arguments 2 ran in 0.09880945999999824 seconds
Function fibonacci with arguments 4 ran in 0.4692909440000079 seconds
Function fibonacci with arguments 1 ran in 6.51999997103303e-07 seconds
Function fibonacci with arguments 0 ran in 1.0500000087176886e-06 seconds
Function fibonacci with arguments 2 ran in 0.11281222700000626 seconds
Function fibonacci with arguments 1 ran in 1.958000012791672e-06 seconds
Function fibonacci with arguments 3 ran in 0.21685028000000273 seconds
Function fibonacci with arguments 5 ran in 0.7868284680000102 seconds
Function fibonacci with arguments 1 ran in 5.699999742402e-07 seconds
Function fibonacci with arguments 0 ran in 1.0729999928571488e-06 seconds
Function fibonacci with arguments 2 ran in 0.11366798399998856 seconds
Function fibonacci with arguments 1 ran in 1.2930000110600304e-06 seconds
Function fibonacci with arguments 3 ran in 0.2176230820000029 seconds
Function fibonacci with arguments 1 ran in 5.839999914769578e-07 seconds
```

Memoization

- Recursion becomes very inefficient if repeated function calls are made
- Can use a built in decorator in order to cache recently calculated values.
- Uses Least Recently Used (LRU) policy

Memoization

```
@functools.lru_cache(maxsize = 100)
#@memoize
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Memoization

- If you use both decorators, you can see the difference between recalculation and accessing a cached value

Memoization

- We can also build our own decorator to store already calculated function values in a dictionary
  - Works, because the dictionary is not garbage collected
  - This simple version works for single argument functions

```
def memoize(function):
    values = {}
    def memoized(arg):
        if arg in values:
            return values[arg]
        else:
            result = function(arg)
            values[arg] = result
        return result
    return memoized
```

Memoization

- The dictionary can become too large, but otherwise, this is a very simple way to speed up recursion that revisits arguments.

Memoization

- Avoiding recursion is often quite a bit faster

```
@timeit
def fib2(n):
    small = 0
    large = 1
    for _ in range(n-1):
        small, large = large, small+large
    return large
```

Memoization

- You could even use an array to store already calculated values

```
@timeit
def fib3(n):
    fibs = [0, 1]
    for i in range(2, n+1):
        fibs.append(fibs[-1]+fibs[-2])
    return fibs[-1]
```