Python Intermediate Programming

Unit 2: Basic Python Algorithms

CHAPTER 6: DECORATORS FOR CALLING FUNCTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- We have discussed decorators before. Typically, we described them as a class that takes an argument that supports some protocol (method calls) and returns an object that supports the same protocol. When the decorator object executes these methods, it performs a bit differently than for the decorated object. The decorator object typically stores a reference to the decorated object and calls it when necessary.

Objectives

- The examples in this lecture, and the `Check_Annotations` class in Programming Assignment #4, use classes to decorate functions by using the `__call__` protocol to decorate/augment how functions are called. Although, some examples don't need classes (and just use functions) to do the decoration.
- Finally, it is appropriate to put this material right after our discussion of recursion and functional programming. Decorating function calls is most interesting when the functions are called many times. And, one call to a recursive function results in it calling itself many times.

Objectives

- We will use the recursive factorial and fib (Fibonacci) functions in our examples below. Both require an argument >= 0.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

def fib(n):
    if    n == 0: return 1
    elif  n == 1: return 1
    else:         return fib(n-1) + fib(n-2)
```

Special Python Syntax for Decorators

LECTURE 1

Special Python Syntax for Decorators

- If Decorator is the name of a decorator (a class or a function) that takes one argument, we can use it to decorate a function object, by writing either

```
def f(params-annotation) -> result-annotation:
    ...
f = Decorator(f)
```

or

```
@Decorator
def f(params-annotation) -> result-annotation:
```

which have the same meanings. `@Decorator` is applied when a module is loaded, as is indicated by the first form.

Special Python Syntax for Decorators

- We can also use multiple decorators on functions. The meaning of

```
@Decorator1
@Decorator2
def f(...)
    ...
```

is equivalent to writing

```
f = Decorator1(Decorator2(f))
```

so Decorator1 decorates the result of `Decorator2` decorating f; the decorators are applied in the reverse of the order they appear (with the closest one to the decorated object applying first).

Special Python Syntax for Decorators

- We have seen a decorator name staticmethod, which we have used to decorate methods defined in classes: those methods that don't have a self parameter and are mostly called in the form **Classname.Staticmethodname(....)**.

Examples of Function Decorators

LECTURE 1

Examples of Function Decorators

- Here are three decorators for functions (three are classes; some can be written easily as functions too).

(1) Track_Calls remembers the function it is decorating and initializes the calls counter to 0; the decorator object overloads the `__call__` method so that all calls to the decorator object increment its calls counter and then actually calls the decorated function (which if recursive, increments the calls counter for every recursive call). Once the function's value is computed and returned, the calls counter instance name can be accessed (via called) and reset (via reset) for tracking further calls.

```
class Track_Calls:
    def __init__(self,f):
        self.f = f
        self.calls = 0
    def __call__(self,*args,**kargs):  # bundle arbitrary arguments to this call
        self.calls += 1
        return self.f(*args,**kargs)   # unbundle arbitrary arguments to call f
    def called(self):
        return self.calls
    def reset(self):
        self.calls = 0
```

So, if we wrote

```
@Track_Calls
def factorial(n):
    if n == 0:
        return 1
    else:
         return n*factorial(n-1)
```

then the name factorial would refer to a Track_Calls object, whose self.f refers to the actual function object factorial defined and whose self.calls is 0.

Examples of Function Decorators

- Examine the picture that shows how a Track_Calls object decorates the factorial function object.
- If we called factorial(3), Python executes the `factorial.__call__(3)` method on the factorial object, which would increment factorial.calls and then call `factorial.f(3)` - the original factorial function object: its body would recursively call `factorial(2)`, which Python executes as `factorial.__call__(2)`.

Examples of Function Decorators

- This process continues, and ultimately `factorial.__call__(3)` returns 6 with factorial.calls = 4. We have seen before that factorial(n) calls itself for n, n-1, n-2, ... 0 for a total of n+1 times. We could write

```
factorial.reset()
print(factorial(10))
print(factorial.called())
```

Examples of Function Decorators

- Examine the picture accompanying this lecture, showing the Fib function (whose body does two recursive calls) and all the recursive calls it generates when called with a variety of numbers. Ignore the colors for now. For example, calling fib(2) does a total of 3 calls to fib (counting itself), fib(3) does a total of 5 calls, fib(4) does a total of 9 calls, fib(5) does a total of 15 calls, fib(6) does a total of 25 calls. We can use Track_Calls to verify these numbers (and compute fib for bigger numbers). For example, fib(10) = 89 does a total of 177 calls, fib(20) = 10,946 and does a total of 21,891 calls.
- Run the program accompanying this lecture to see the value (and number of function calls) needed to evaluate fib(0) through fib(30). Notice that the last few calculations take a noticeable amount of time: computing fib(30) requires 2,692,537 function calls!

Examples of Function Decorators

- We can write this decorator as the following function instead

```
def track_calls(f):
    def call(*args,**kargs):
        call.calls += 1
        return f(*args,**kargs)

    call.calls = 0 # define calls attribute on call function-object!
    return call
```

Examples of Function Decorators

- Here we define an inner-function call, which is returned by track_calls. Before returning this function object, a calls attribute is defined for that object and initialized to 0; inside the call function that instance name is incremented before the original function (f) is called and the value it computes returned.
- We know objects have namespaces stored in `__dict__` of the object, and function objects are just a kind of objects. After executing

```
factorial = track_calls(factorial)
```

- We can examine and rebind `fraction.calls`: e.g, print(fraction.calls) and `fraction.calls = 0`.

Examples of Function Decorators

- In fact, we can define the track_calls function below, so that we can call the methods called/reset on it. Here we bind the attributes reset/called to functions (on named - because it executes a statement - and one lambda, because it just returns a value)

```
def track_calls(f):
    def call(*args,**kargs):
        call.calls += 1
        return f(*args,**kargs)
    call.calls = 0                  # define calls, reset, and called attributes on
                                    # call function-object; bind the first to a
    def reset(): call.calls = 0     # data object, the next two to function objects
    call.reset = reset
    call.called = lambda : call.calls
    return call
```

Examples of Function Decorators

- In this case we can write (exactly as we did for the Track_Calls class:

```
factorial.reset()
print(factorial(10))
print(factorial.called())
```

- I will continue to show equivalent class/function definitions for the decorators described below, but in a simplified form (like the original track_calls above).
- At the end of this lecture, I will briefly explain the reason for using classes instead of functions: the ability to overload the `__getattr__` function for using multiple decorators at the same time.

Examples of Function Decorators

- (2) Memorize remembers the function it is decorating and initializes a dict to {}. It will use this dict to cache (keep track of and be able to access quickly) the arguments to calls and the value ultimately returned by the function. The decorator object overloads the `__call__` method so that all calls to the decorator object first check to see if the arguments are already cached in the dict (if so, its value is there too), and if so their associated value is returned immediately, without executing the code in the decorated function; if not the decorated function is called, its answer is cached in the dict with the function's arguments, and the answer is returned.
- For simplicity here, I'm assuming all arguments are positional (so no **kargs). Also, since the arguments are used as keys in a dictionary, they must be immutable/hashable (which ints are). If the weren't, we could try to convert each argument an immutable one for use in the cache dict: e.g,, convert a list into an equivalent tuple.

Examples of Function Decorators

- In this way, a function never has to compute the same value twice. Memoization is useful for multiply-recursive calls, as in the fibonacci function (not so much in factorial, where it computes each value only once).

```
class Memoize:
    def __init__(self,f):
        self.f = f
        self.cache = {}

    def __call__(self,*args):
        if args in self.cache:
            return self.cache[args]
        else:
            answer = self.f(*args)
            self.cache[args] = answer
          return answer

    def reset_cache(self):
        self.cache = {}
```

Examples of Function Decorators

- Examine the picture accompanying this lecture, showing the Fibonacci function. When decorated by Memorize, the only calls that actually compute a result are those in green. As each green function calls finishes, it caches its result in the dictionary. Afterward, calling fib with that argument again will obtain the result from the cache immediately (obviating the need for all the calls in white). With memorization, calling fib(n) does a total of n+1 function calls.
- Run the program accompanying this lecture to see the value (and number of function calls) needed to evaluate fib(0) through fib(30). This time, uncomment **@Memorize** and the `fib.reset_cache()` call in the loop. Notice that all the calculations are done instantaneously: with memorization, computing fib(30) requires only 31 function calls (with cached values computed immediately billions of times).

Examples of Function Decorators

- We can also write memorize as a function, to return a wrapper function (it can be named anything) that does the same operations as the class above. Note that cache, a name local to memorize, is used in wrapper but not accessible outside wrapper (unlike what we did with call.calls above)

```
def memoize(f):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        else:
            answer = f(*args)
            cache[args] = answer
            return answer
    return wrapper
```

Examples of Function Decorators

(3) Illustrate_Recursive remembers the function it is decorating and initializes a tracing variable to False. The decorator object overloads the `__call__` method so that all calls to the decorator object just return the result of calling the decorated function (if tracing is off). Calling .illustrate(...) on the decorator calls the illustrate method, which sets up for tracing, and then uses `__call__` to trace all entrances and exists to the decorated function printing indented/outdented information for each function call/return.

```
class Illustrate_Recursive:
    def __init__(self,f):
        self.f = f
        self.trace = False

    def illustrate(self,*args,**kargs):
        self.indent = 0
        self.trace = True
        answer = self.__call__(*args,**kargs)
        self.trace = False
        return answer

    def __call__(self,*args,**kargs):
        if self.trace:
            if self.indent == 0:
                print('Starting recursive illustration'+30*'-')
            print (self.indent*"."+"calling", self.f.__name__+str(args)+str(kargs))
            self.indent += 2
        answer = self.f(*args,**kargs)
        if self.trace:
            self.indent -= 2
            print (self.indent*"."+self.f.__name__+str(args)+str(kargs)+" returns", answer)
            if self.indent == 0:
                print('Ending recursive illustration'+30*'-')
        return answer
```

Examples of Function Decorators

- Run the program accompanying this lecture to example of this trace (and others by changing the program. Here is what the program prints for a call to factorial and fibonacci.

```
@Illustrate_Recursive
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Examples of Function Decorators

```
Starting recursive illustration------------------------------
calling factorial(5,){}
..calling factorial(4,){}
....calling factorial(3,){}
......calling factorial(2,){}
........calling factorial(1,){}
..........calling factorial(0,){}
..........factorial(0,){} returns 1
........factorial(1,){} returns 1
......factorial(2,){} returns 2
....factorial(3,){} returns 6
..factorial(4,){} returns 24
factorial(5,){} returns 120
Ending recursive illustration--------------------------------
```

Examples of Function Decorators

- Factorial is a linear recursive function (one recursive call in its body) so the structure of its recursion is simple. Each factorial calls the one below it (indented); when the bottom one returns 1 for its base case, each call above it (un-indented) can compute and return its result.

```
@Illustrate_Recursive
def fib(n):
    if    n == 0: return 1
    elif  n == 1: return 1
    else:         return fib(n-1) + fib(n-2)
print(fib.illustrate(5))
```

```
Starting recursive illustration------------------------------
calling fib(5,){}
..calling fib(4,){}
....calling fib(3,){}
......calling fib(2,){}
........calling fib(1,){}
........fib(1,){} returns 1
........calling fib(0,){}
........fib(0,){} returns 1
......fib(2,){} returns 2
......calling fib(1,){}
......fib(1,){} returns 1
....fib(3,){} returns 3
....calling fib(2,){}
......calling fib(1,){}
......fib(1,){} returns 1
......calling fib(0,){}
......fib(0,){} returns 1
....fib(2,){} returns 2
..fib(4,){} returns 5
..calling fib(3,){}
....calling fib(2,){}
......calling fib(1,){}
......fib(1,){} returns 1
......calling fib(0,){}
......fib(0,){} returns 1
....fib(2,){} returns 2
....calling fib(1,){}
....fib(1,){} returns 1
..fib(3,){} returns 3
fib(5,){} returns 8
Ending recursive illustration------------------------------
```

Examples of Function Decorators

- The fib function is NOT a linear recursive function: its body contains two recursive calls. So the structure of its recursion is more complicated. This is why the fib function is a good one to test both Track_Calls and Memoize. Even for fairly small arguments (under 30), it produces a tremendous number of calls and can be sped-up tremendously by memoizing it.
- The mns function in the previous lecture (minimum number of stamps) does even more calls: one for each denomination. It too can be vastly sped-up by using memoize: see the modules whose names end in "fast" in the stamps download; run their code to see completely computed results.

Delegation of attribute lookup

LECTURE 1

Delegation of attribute lookup

- When using (multiple) decorators, we need a way to translate attribute accesses on the decorator into attribute accesses on the decorated. There is no simple mechanism to do this with functions, but it easy to do with classes: by overloading the `__getattr__` method as follows (which should be one to all three classes above).

```
def __getattr__(self,attr):  # if attr not here, try self.f
    return getattr(self.f,attr)
```

- So, if we use the following two decorators

```
@Track_Calls
@Illustrate_Recursive
def fib(....):
    ...
```

Delegation of attribute lookup

- fib is a Track_Calls object, whose .f attribute is an Illustrate_Recursive object whose .f attribute is the actual fib function. If we then wrote fib.illustrate(5) Python would try to find the illustrate attribute of the Track_Calls object; there is no such attribute there, so it fails and then calls the `__getattr__` of the Track_Calls class, which "translates" the failed attribute access into getting the same attribute from the .f object (from the decorated class object, an object of the Illustrate_Recursive class, which does define such an attribute, as a method which can be called).
- Generally, this is called delegation: where an "outer" object that does not have some attribute delegates the attribute reference to an inner object. Decorators often use exactly this form of delegation, so the decorator object can process its attributes and delegate lookup to all the attributes of the decorated object.

Delegation of attribute lookup

- Now we will study an interesting combination of using decorators and the functools.partial function (discussed in the previous lecture). Let's look at the following simple decorator, which takes a function and its name as arguments; every time the function is called the decorator prints the function's name and the result it computes.

```
class Trace:
    def __init__(self,f,f_name):
        self.f = f
        self.f_name = f_name

    def __call__(self,*args,**kargs):
        result = self.f(*args,**kargs)
        print(self.f_name+'called: '+str(result))
        return result
```

Delegation of attribute lookup

- Now, suppose we want to decorate a function f defined simple as

```
def f(x):
    return 2*x
```

- If we write

```
@Trace
def f(x):
    return 2*x
```

Delegation of attribute lookup

- Python raises a **TypeError** exception, because the __init__ for Trace requires two arguments, not one. We can avoid the @ form of decorators and write

```
def f(x):
    return 2*x
f = Trace(f,'f')
```

to solve the problem, but without using the @Decorator form. Can we do something to use this form? The problem is that we have two arguments to __init__ but @Decorator requires just one, so we can use functools.partial to pre-supply the second argument.

- We can write

```
f_Trace = functools.partial(Trace,f_name='f')
@f_Trace
def f(x):
    return 2*x
```

Delegation of attribute lookup

- But even that is a bit clunky, because we are defining the name f_Trace but using it only once (we are not likely to trace other functions named `f`). We don't need this name; instead, we can write

```
@functools.partial(Trace,f_name='f')
def f(x):
    return 2*x
```

directly using the result returned from partial as the decorator. This allows us to use the standard @**Decorator** form (with possibly more than one decorator).

- Now calling f(1) would return the result 2 and cause Python to print

```
f called: 2
```