

DECORATORS

Lets review what we ended with last time: Classes are Objects
This means they can be passed around, created at runtime
and as we saw, we can "add" features to existing classes
without modifying the code of the class:

```
def counted(C):  
    class CC(C):  
        idx = 0  
        def __init__(self, *args):  
            try: C.__init__(self, *args)  
            except: pass  
            CC.idx += 1  
            self.inst_idx = CC.idx  
    return CC
```

This function takes as an argument a class (C) and returns a derived / subclass CC.

The subclass has all the attributes of C and then some ...

The derived class CC has a class variable / field / attribute called idx. Remember, CC is a namespace, so CC.idx is the attribute idx belonging to the namespace of the class CC.

The __init__ method of the derived class overrides the __init__ of the superclass C: it first calls the init of the superclass. If no such method exists, then an exception gets thrown and gets caught. Finally, after the instance object has been initialized as before, we increment the class counter CC.idx and use it to "stamp" the

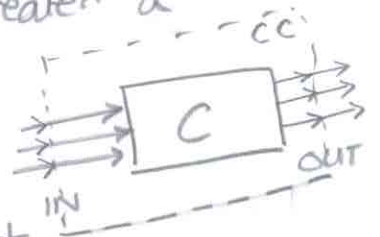
Instance created by writing to a new field `inst_idx`.
Thus, `CC.idx` keeps track of how many instances have been created, and each instance has an extra attribute which tells us how many instances were created prior to that one!

Note, ① that other than this extra attribute, the instances will behave exactly as they did before!, and ② this works for pretty much any class. Let's try it out on some of the classes we have seen before.

```
>>> CPoint = counted(Point)
>>> p1 = CPoint(0,0)
>>> p2 = CPoint(1,10)
>>> p3 = CPoint(100,100)
>>> p4 = CPoint(25,45)
>>> p2.inst_idx      } # 2nd Instance
2
>>> p3.inst_idx      } # 3rd Instance
3
>>> CPoint.idx        } # total number of
4                     } inst. so far
```

```
x >>> CCounter = counted(Counter)
>>> c1 = CCounter(10)
>>> c2 = CCounter(0)
>>> c3 = CCounter(-1992)
>>> c4 = CCounter(1000)
>>> c5 = CCounter(10)
>>> for i in range(10):
    print c1.next(), c2.next(), c3.next(), c4.next(), c5.next()
>>> c4.inst_idx
4
```

So that's pretty cool! We can "extend" the behavior of ANY class like this. What we did intuitively is create a WRAPPER around the original class:



The wrapper gets to intercept all incoming calls, can call the original "wrapped" object and observe its outputs and return them to the external world. So, can add behaviors without messing with the inner box.

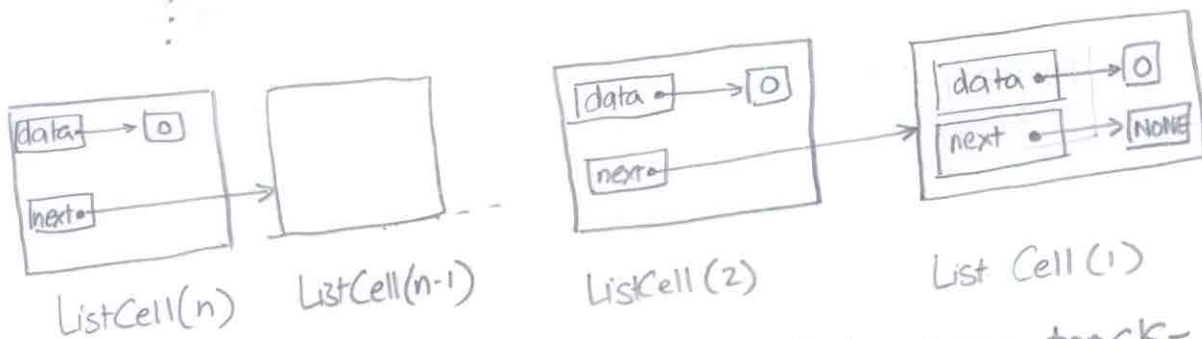
But, before we go on, can anyone see a problem with this approach?

- ① All the remaining code, using Point, Counter etc. must change to use CPoint / CCounter.
- ② What if the class referred to itself like so:

```
class ListCell:
```

```
    def __init__(self, n):
        self.data = 0
        if n > 1: self.next = ListCell(n-1)
        else: self.next = None
```

} creates a list of n cells, each with a data field initialized to 0



So, let's try using "counted list cells" to keep track of how many cells have been created.

```
>>> CListCell = counted(ListCell)
>>> L = list(10)
```

```

>>> l.data = 1
>>> l.next.data = 2
>>> l.next.next.data = 3
>>> str(l)

```

} It's a bona fide list with 10 cells, however...

```

>>> ListCell.idx

```

1

Uh! Because? Only the "top-level" `--init--` got wrapped.
 Why? Because the `--init--` inside `ListCell` calls `ListCell.--init--(n-1)` recursively, not `ListCell.--init--` which is where the wrapping happens. How to solve this problem? Well, it's all about the name. If `ListCell.--init--(n-1)` is the recursive call, we have to make sure that it refers to the wrapped class like

So:

```

>>> ListCell = counted(ListCell)

```

Says ① take the "original" class that the name ListCell refers to,

② create a wrapped class around it,

③ make `ListCell` refer to the "wrapped" class.

Thus, when executing the recursive call `ListCell.--init--(n-1)` we now execute the "wrapped" init. Cool.

```

>>> l = ListCell(10)

```

```

>>> str(l)

```

```

>>> m = ListCell(50)

```

```

>>> ListCell.idx

```

60

} 60 cells created so far...

```

>>> m.inst.idx

```

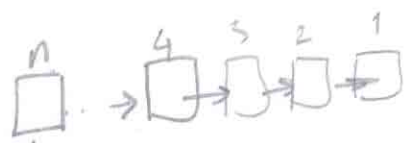
60

```

>>> m.next.mst.idx

```

59

} Each cell is stamped! 
 Note the lists are created starting ~~at~~ the end, ... Hence `m.mst.idx` is 60

We just wrapped or "decorated" the classes Point, Counter, ListCell using the function counted. Python has a notation for this: they are called "Decorators"; and you write them as:

@ counted
class Point:



⇓ Equiv to

class Point:



Point = counted(Point)

@ counted
class Counter



⇓ Equiv to

class Counter:



Counter = counted(Counter)

@ counted
class ListCell



⇓ Equiv to

class ListCell:



ListCell = counted(ListCell)

In general, we can write:

@ decorator
definition d

which is exactly the same as:

definition d
d = decorator(d)

decorator: Can be anything that is "callable" i.e. takes an argument. So either a function object, or a class object (whose initializer gets called when you do decorator(d))

definition d: Can be any definition — either a function definition:

def d(...):
FUN-BODY

or a class definition:

class d(...):
CLASS-BODY

So, that's very flexible. We saw an example where a class was decorated with a function. (counted) ⑥
(Point, Counter, ListCell)

In general, the decorator should "return" the same type of object as its argument.

Class	→	Class
Function	→	Function

Let's see how a function can be decorated.

So, the standard way of debugging is "printf" debugging where you have to add printf statements that, say, log all calls made, the arguments, and the values returned by the call. This requires the tedious addition of print statements all over your code, which print out args/return values, which clutters code and makes it look ugly!

We need a wrapper that ① intercepts args, prints them, ② calls the real function ③ captures and prints the return value, and then returns it:

```
>>> def logged(f):  
    def g(*args):  
        print "Call ", f.__name__, "args:", str(args)  } # ①  
        rv = f(*args)                                     } # ②  
        print "Returns", str(rv)                         } # ③  
        return rv  
    return g
```

Thus, for any Python function f , $\text{logged}(f)$ is a wrapped version that prints arguments and outputs...

Lets take it out for a spin:

```
>>> @logged
def linecount(...):
```

- equivalent to

```
def linecount(...):
    |
    linecount = logged(linecount)
```

```
>>> @logged
def fac(n):
```

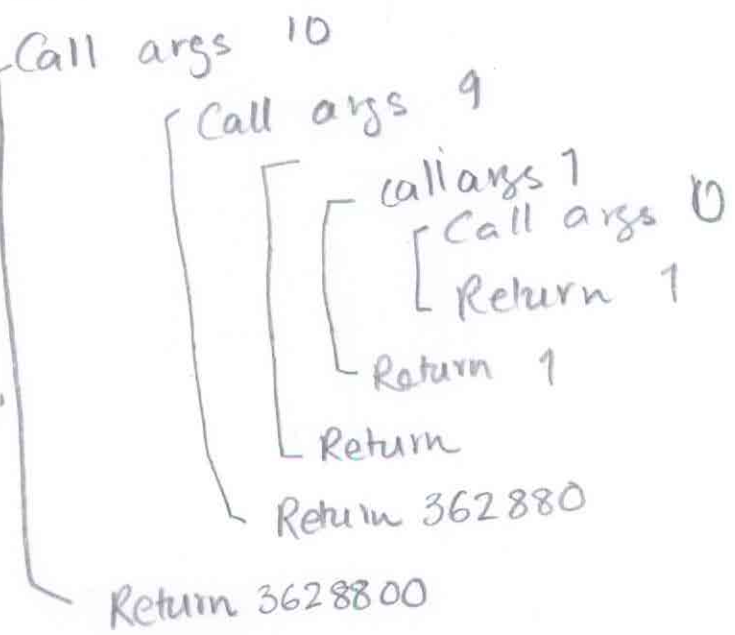
- equiv to

```
def fac(...):
    |
    fac = logged(fac)
```

```
>>> x = linecount("lec25.py")
Call linecount args lec25.py
Returns . 212
```

```
>>> x
212
>>> y = fac(10)
Call fac args : 10
" " " : 9
Returns : 0
" " " : 1
Returns : 1
```

Because each call is wrapped we get the calls, Returns Nested. Not pretty. We'd like



Here, the wrapper needs to "know" the nesting depth... hmmm.
Part of PA4.

To do this, the wrapper needs some kind of "memory". It has to "remember" what the current "size" of the call stack is and increment the size when it sees a call, and decrement when it gets a return.

Let's do a simpler one. Profiling: Suppose you would like to "count" how many times a function gets called during the course of execution. Again, you could have a "global variable" for each function and increment it inside the body of the function but that is tedious and ugly.

We can try to use a decorator, but there is a difference from the previous example - now the wrapper must have some memory - it needs to count how many times the function has been called. How can we do this? Wrapped "thing" needs

- ① "Memory" / "Data" to store number of times called
- ② To be callable - so that it "behaves" exactly like the original function.

- | | |
|-------------|--------------------------|
| ① data | } Sounds like an OBJECT! |
| ② operation | |

Recall that in Python, a function is an object too! It's an object which has a method "--call--" so if x is an object with a method --call-- then Python translates $x(...)$ to $x.--call--(...)$

So now, we can build a wrapped object whose --call-- method just calls the actual function, and which has an attribute storing the number of times it has been called!

Inherits / derived from

```
>>> class Profiled(object):
    def __init__(self, f):
        self.count = 0
        self.__f = f
        self.__name__ = f.__name__
    def __call__(self, *args):
        self.count += 1
        return self.__f(*args)
```

} initializes count to 0
} initialized to "f"
} the wrapped function
} increment count
} call wrapped function

Now, remember that `Profiled(f)` returns a new instance of class `Profiled`. This instance has a `--call--` method that just returns what `f` would have with the same args, as it calls the function stored in the `--f` field, which is just "f".

However, the instance also has a counter that stores how many times the `--call--` method is invoked, i.e. how many times "f" is called.

So, we can say:

```
>>> @Profiled
def fac():
    |
```

} equiv to `def fac():`
 |
 `fac = Profiled(fac)`

```
or >>> @Profiled
def linecount(...):
    |
```

} equiv to `def linecount():`
 |
 `linecount = Profiled(linecount)`

Again, by "rebinding" the name "fac" the recursive calls all refer to the wrapper i.e. decorated object so `fac(n-1)` gets translated to `fac.--call--(n-1)`

```

>>> fac(10)    } calls fac exactly
                  } as before
>>> fac.count   } tells you how many
11               } calls made into fac!

>>> fac(5)
120
>>> fac.count
17

```

Again, as before it works for any function.

```

@Profiled | @Profiled | @Profiled
def fac   | def linecount | def kthline ...

```

What is more, you can "compose" the decorators:

```

@Profiled
@logged
def fac

```

Same as:

```

def fac():
    ...
fac = logged(fac)  ①
fac = Profiled(fac) ②

```

```

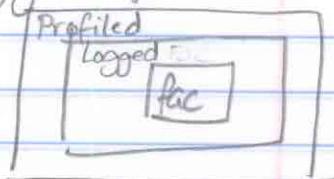
>>> fac(10)
:
:
>> fac(5)
:
>>> fac.count
17

```

logging

logging

How? Well, the `@logged` created a new wrapped version of the function that did the logging, and `@Profiled` added another wrapper around the logged-fac



Each layer calls the inner layer so ^{orig} functionality is preserved.
but new features added by the layer!

Note that in this case the ORDER of wrapping/decorating matters. Try to figure out why...

Lets see one last example. Suppose you want a wrapper to check if the arguments passed into a function were of the right type. If so, you proceed with the function. If not, you raise an exception, without calling the function - thus "shielding" the function from the wrong kinds of arguments

```
>>> def checkargtype(i, t):  
    def decorator(f):  
        def decorated(*args):  
            if not (isinstance(args[i], t)):  
                raise Exception "..."  
            decorated.__name__ = f.__name__  
            return decorated  
        return decorator
```

checkargtype takes a pair (i, t) and returns a new function decorator. Think of currying as in ML.

This new function takes a function "f" and returns a "decorated" or wrapped version of f where we first intercept the args and checks if the i^{th} arg is indeed an instance of type t (i.e the i^{th} arg has type t) if so, the wrapped function f is called...

So now we write

```
>>> @check_argtype(0, str)
def linecount(...):
```



This is the same as:

```
>>> def linecount(...):
```



```
    linecount = check_argtype(0, str)(linecount)
```

this is itself a function
"curried"-style params

```
>>> linecount("lec25.py")
```

```
>>> linecount(4+2)
Exception
```

} catches error early,

Again, we can compose the decorators to check several args.

```
>>> @check_argtype(1, int)
    @check_argtype(0, str)
    def kthline(...):
```

Two layers of wrapping

Outer layer checks arg 1
calls inner

Inner layer checks arg 0
calls function

```
>>> kthline("jhala", 10)
```

} Both args bad but
outer layer catches arg 1
first and that's the
exception thrown.

This is a case of a more general programming technique to decorate objects and smoothly, cleanly add features and behaviors at runtime while maintaining original properties. Called "Decorator Design Pattern". See more in a fun reading posted on resources page.