DECORATORS

Let's 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:

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

```python
>>> 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
inst. so far
4

>>> 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.mst_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
```

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

```python
>>> CListCell = counted(ListCell)
>>> l = list(10)
```
1. data = 1
2. next.data = 2
3. next.next.data = 3

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

```python
>>> ClistCell.idx

1
```

Uh! Because? Only the "top-level" __init__ got wrapped.
Why? Because the __init__ inside ListCell calls ListCell.__init__(n-1) recursively, not ClistCell.__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:

```python
>>> 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.

```python
>>> l = ListCell(10)
>>> str(l)
>>> m = ListCell(50)
>>> ListCell.idx

60
```

```python
>>> m.inst.idx

60
```

```python
>>> m.next.inst.idx

Each cell is stamped!
```

```plaintext
Note: The lists are created starting at the end... Hence m.inst.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:

```python
@counted
class Point:

Point = counted(Point)
```

```python
@counted
class Counter:

Counter = counted(Counter)
```

```python
@counted
class ListCell:

ListCell = counted(ListCell)
```

In general, we can write:

```python
@decorator
definition d
```

which is exactly the same as:

```python
definition d
d = decorator(d)
```

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

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

```python
def d(...):
    [FUN-BODY]
```

or a class definition:

```python
class d(...):
    [CLASS-BODY]
```
So, that's very flexible. We saw an example where a class was decorated with a function.
(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:

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

Thus, for any Python function f, logged(f) is a wrapped version that prints arguments and outputs...
Let's take it out for a spin:

```python
def linecount(\_\_\_):
    return equivalence
    linecount = logged(linecount)

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

Recall that in Python, a function is an object too! Its 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.
```python
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)
```

Inherits/derived from

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:

```python
@Profiled
def fac(n):
    return fac(n-1)
```

Equivalent to:

```python
fac = Profiled(fac)
```

or

```python
@Profiled
def linecount(...):
    return linecount(...)
```

Equivalent to:

```python
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
     3 as before
>>> fac.count  # tells you how many
     calls made into fac!

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

Again, as before it works for any function.

@Profied | @Profied | @Profied
def fac   | def linecount | def kthline ...

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

@Profied
@logged
def fac:  # same as:
    fac = Logged(fac)  # 1

How? Well, the @logged
created a new wrapped version
of the function that did
the logging, and @Profied
added another wrapper around
the logged fac
Each layer calls the inner layer so 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...

Let’s 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.

```python
>>> 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 had type `t` if so, the wrapped function `f` is called….
So now we write

```
@checkargtype (0, str)

def linecount ():
    # linecount = checkargtype (0, str) (linecount)
    this is itself a function
    "curried"-style params
```

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

```python
>>> linecount (4+2)  # catches error 'early
Exception
```

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

```python
@checkargtype (1, int)
@checkargtype (0, str)

def kthline (...):
    # Two layers of wrapping
    # Outer layer checks arg 1
    # Inner layer checks arg 0
    # calls function

>>> kthline ("jhalo", 10)  # Both args bad but
outer layer catches arg 1
first and then 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.