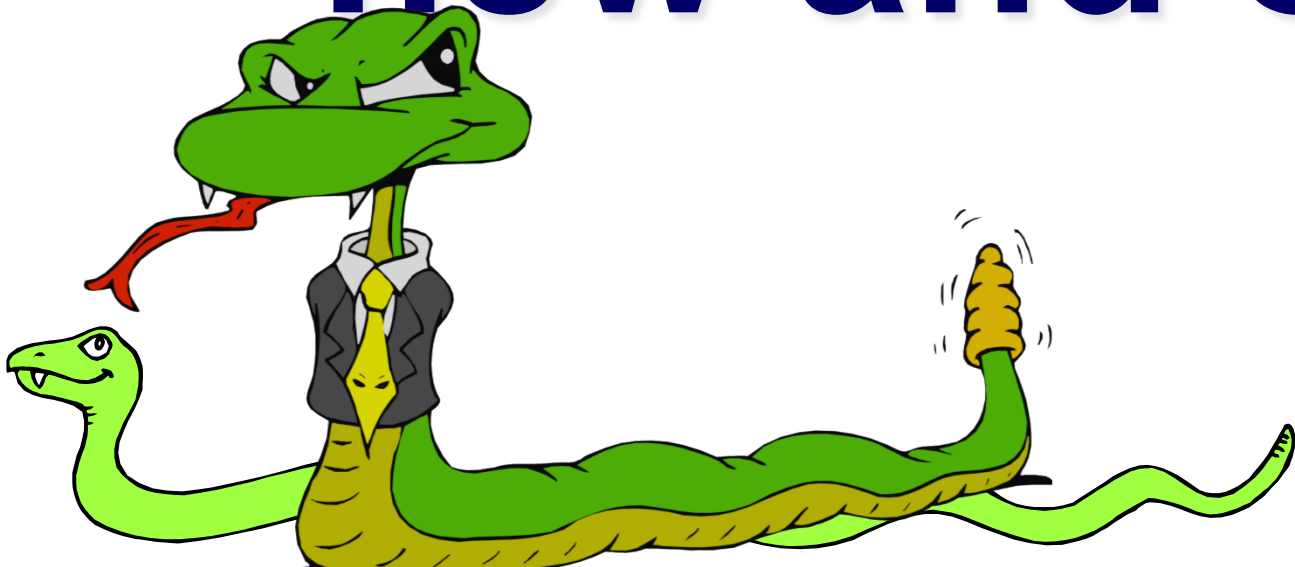


# Python classes: new and old



# New and classic classes



- With Python 2.2, classes and instances come in two flavors: old and new
- New classes cleaned up the language by
  - Unifying classes and types
  - Allowing all built-in types to be sub-classed
- For compatibility, default 2.x class is old style; Python 3 only has new classes
- New classes support interesting features
- New classes subclass object

# New class *student*

```
class Student(object):  
    """A new class representing a student"""  
    def __init__(self, n, a):  
        self.full_name = n  
        self.age = a  
    def get_age(self):  
        return self.age
```

# Class property

- One neat feature in new classes is the [property](#) function
- It's a better way to manage private attributes, and getter and setter methods
- While still keeping access simple
- We'll also see [decorators](#), an interesting feature

# Boxes, little boxes



```
class Box(object):
    def __repr__(self):
        return "<A box with length:%s, width:%s, area:%s>"
        % (self.length, self.width, self.area)
```

```
class Box1(Box):
    """A rectangle"""
    def __init__(self, l=1, w=1):
        self.length = l
        self.width = w
        self.area = l * w
```

# Boxes, little boxes



- Ok, but not perfect

```
>>> from box import *
```

```
>>> b1 = Box1(2, 3)
```

```
>>> b1
```

```
A box with length:2, width:3, area:6
```

```
>>> b1.area = 9
```

```
>>> b1
```

```
<A box with length:2, width:3, area:9>
```

# Boxes, little boxes



- Let's use a getter method for area

```
class Box2(Box):
```

```
    """A rectangle with area getter"""
```

```
    def __init__(self, l=1, w=1):
```

```
        self.length = l
```

```
        self.width = w
```

```
    def get_area(self):
```

```
        return self.length * self.width
```

```
    def set_area(self, val=1):
```

```
        print "Warning: area is read only!"
```

# Boxes, little boxes



- Not without problems, though...

```
>>> from box import *
```

```
>>> b2 = Box2(2, 3)
```

```
>>> b2
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
File "box.py", line 5, in __repr__
```

```
    return "<A box with length:%s, width:%s, area:%s>" %  
        (self.length, self.width, self.area)
```

AttributeError: 'Box2' object has no attribute 'area'



# Boxes, little boxes



- There are two problems that the getter/setter idiom creates
- It results in different access patterns for different attributes ☹️
- If we start with plain attributes and later decide to use getter and setters, we may have a lot of code to change ☹️

# Boxes, little boxes



- Python solution: [property\(\)](#)

```
class Box3(Box):
```

```
    """A rectangle with area property"""
```

```
    def __init__(self, l=1, w=1):
```

```
        self.length = l
```

```
        self.width = w
```

```
    def get_area(self): return self.length * self.width
```

```
    def set_area(self, val=1):
```

```
        print "Warning: area is read only!"
```

```
    area = property(get_area, set_area)
```

# Boxes, little boxes



- The `property()` function takes optional args for an attribute's getter, setter, deleter and doc string  
`property([fget[, fset[, fdel[, doc]]]])`
- and returns an object
- Not providing a setter results in a read-only attribute

# Decorated Boxes



- Use Python [decorators](#)

```
class Box4(Box):
```

```
    """A rectangle with area property"""
```

```
    def __init__(self, l=1, w=1):
```

```
        self.length = l
```

```
        self.width = w
```

```
    @property
```

```
    def area(self): return self.length * self.width
```

```
    @area.setter
```

```
    def area(self, val=1): print "Warning: area is read only!"
```

<http://cs.umbc.edu/courses/331/current/code/python/box.py>

# Python Decorators



- Python's decorator is syntactic sugar  
@foo  
def bar (x): pass
- Is the equivalent of  
def bar (x): pass  
bar = foo(bar)
- That is: rebind the name bar to the result of calling foo with the function object bar
- foo typically returns a modified version of the function bar

# Decorator example: trace



```
def trace(f):
    def new_f(*args)
        print 'Entering %s%s' % (f.__name__, args)
        result = f(*args, **kwargs)
        print 'Exiting %s%s with %s' % (f.__name__, args, result)
        return result
    return new_f
```

@trace

```
def sum(n, m):
    return n + m
```

```
>>> sum(10,20)
Entering sum(10, 20)
Exiting sum(10, 20) with 30
30
```

# Decorator example: trace



```
@trace
```

```
def fact(n): return 1 if n<2 else n * fact(n-1)
```

```
>>> fact(4)
```

```
Entering fact(4,)
```

```
Entering fact(3,)
```

```
Entering fact(2,)
```

```
Entering fact(1,)
```

```
Exiting fact(1,) with 1
```

```
Exiting fact(2,) with 2
```

```
Exiting fact(3,) with 6
```

```
Exiting fact(4,) with 24
```

```
24
```

# Decorated Boxes



```
class Box5(Box):
    def __init__(self,l=1,w=1):
        self.length = l
        self.width = w
        self._color = None

    @property
    def area(self): return self.length * self.width

    @area.setter
    def area(self, val=1): self.length = self.width = math.sqrt(val)

    @property
    def color(self): return self._color

    @color.setter
    def color(self, val): self._color = val

    @color.deleter
    def color(self): del self._color
```