Attribute Access in Classes and Instances

getattr() and setattr()

These built-in functions allow attribute access through string arguments.

```
class This(object):       # a simple class with one class variable
    a = 5

x = This()

print getattr(x, 'a')     # 5:  finds the 'a' attribute in the class

setattr(x, 'b', 10)       #     set x.b = 10 in the instance

print x.b                 # 10: retrieve x.b from the instance
```

Similar to the dict method **get()**, a default argument can be passed to **getattr()** to return a value if the attribute doesn't exist (otherwise, a missing attribute will raise an **AttributeError** exception).

```
class InstMake(object):        # create a featureless class so we can
                               # play with its instance
    pass

x = InstMake()

curr_val = getattr(x, 'intval', 0)   # no 'intval' attribute, so default 0
setattr(x, 'intval', 10)             # set 'intval' to 10

print x.__dict__                     # {'intval': 10}
```

We might want to use these functions as a dispatch utility: if our program is working with a string value that is the name of an attribute, we can use the string directly.

```
var = 'hElLo'
for methodname in ('upper', 'lower', 'title'):
    print getattr(var, methodname)()           # call each method through the attribute
                                               # HELLO
                                               # hello
                                               # Hello
```

Special methods __getattr__, __getattribute__ and __setattr__

These special methods are called when we access an attribute (setting or getting). We can implement them for broad control over our custom class' attributes.

```
class MyClass(object):

    classval = 5

    # when read attribute does not exist
    def __getattr__(self, name):
        default = 0
        print 'getattr: "{}" not found; setting default'.format(name)
        setattr(self, name, default)
        return default

    # when attribute is read
    def __getattribute__(self, name):
        print 'getattribute:  attempting to access "{}"'.format(name)
        return object.__getattribute__(self, name)

    # when attribute is assigned
    def __setattr__(self, name, value):
        print 'setattr:  setting "{}" to value "{}"'.format(name, value)
        self.__dict__[name] = value

x = MyClass()

x = MyClass()

x.a = 5             # setattr:  setting "a" to value "5"

print x.a           # getattribute:  attempting to access "__dict__"
                    # getattribute:  attempting to access "a"
                    # 5

print x.ccc         # getattribute:  attempting to access "ccc"
                    # getattr: "ccc" not found; setting default
                    # setattr:  setting "ccc" to value "0"
                    # getattribute:  attempting to access "__dict__"
                    # 0
```

__getattribute__: implicit call upon attribute read. Anytime we attempt to access an attribute, Python calls this method if it is implemented in the class.

__getattr__: implicit call for non-existent attribute. If an attribute does not exist, Python calls this method -- regardless of whether it called **__getattribute__**.

recursion alert: we must use alternate means of getting or setting attributes lest Python call these methods repeatedly:
__getattribute__(): use **object.__getattribute__(self, name)**
__setattr__(): use **self.__dict__[name] = value**

e.g., use of **self.attr = val** in **__setattr__** would cause the method to call itself.

@property: attribute control

This *decorator* allows behavior control when an individual attribute is accessed, through separate @property, @setter and @deleter methods.

```
class GetSet(object):

    def __init__(self,value):
        self.attrval = value

    @property
    def var(self):
        print 'getting the "var" attribute'
        return self.attrval

    @var.setter
    def var(self, value):
        print 'setting the "var" attribute'
        self.attrval = value

    @var.deleter
    def var(self):
        print 'deleting the "var" attribute'
        self.attrval = None

me = GetSet(5)

me.var = 1000    # setting the "var" attribute
print me.var     # getting the "var" attribute
                 # 1000

del me.var       # deleting the "var" attribute
print me.var     # getting the "var" attribute
                 # None
```

Note that each decorated method is called **def var**. This would cause conflicts if it weren't for the decorators.

One caveat: since the interface for attribute access appears very simple, it can be misleading to attach computationally expensive operations to an attribute decorated with @property.

Attribute access: descriptors

A descriptor is an attribute that is linked to a separate class that defines **__get__()**, **__set__()** or **__delete__()**.

```
class RevealAccess(object):
    """ A data descriptor that sets and returns values and prints a message declaring access. """

    def __init__(self, initval=None):
        self.val = initval

    def __get__(self, obj, objtype):
        print 'Getting attribute from object', obj
        print '...and doing some related operation that should take place at this time'
        return self.val

    def __set__(self, obj, val):
        print 'Setting attribute from object', obj
        print '...and doing some related operation that should take place at this time'
        self.val = val


# the class we will work with directly
class MyClass(object):
    """ A simple class with a class variable as descriptor """
    def __init__(self):
        print 'initializing object ', self

    x = RevealAccess(initval=0)  # attach a descriptor to class attribute 'x'


mm = MyClass()                   # initializing object  <__main__.MyClass object at 0x10066f7d0>

mm.x = 5                         # Setting attribute from object <__main__.MyClass object at 0x1004de910>
                                 # ...and doing some related operation that should take place at this time

val = mm.x                       # Getting attribute from object <__main__.MyClass object at 0x1004de910>
                                 # ...and doing some related operation that should take place at this time

print 'retrieved value: ', val   # retrieved value:  5
```

You may observe that descriptors behave very much like the **@property** decorator. And it's no coincidence: **@property** is implemented using descriptors.

__slots__

This class variable causes object attributes to be stored in a specially designated space rather than in a dictionary (as is customary).

```
class MyClass(object):
    __slots__ = ['var', 'var2', 'var3']

a = MyClass()

a.var = 5
a.var2 = 10
a.var3 = 20
a.var4 = 40   # AttributeError:  'MyClass' object has no attribute 'var4'
```

All objects store attributes in a designated dictionary under the attribute __dict__. This takes up a fairly large amount of memory space for each object created.

__slots__, initialized as a list and as a class variable, causes Python *not* to create an object dictionary; instead, just enough memory needed for the attributes is allocated.

When many instances are being created a marked improvement in performance is possible. The hotel rating website **Oyster.com** reported a problem solution (http://tech.oyster.com/save-ram-with-python-slots/) in which they reduced their memory consumption by a third by using __**slots**__.

<u>Please note however</u> that slots should not be used to limit the creation of attributes. This kind of control is considered "un-pythonic" in the sense that privacy and control are mostly cooperative schemes -- a user of your code should understand the interface and not attempt to subvert it by establishing unexpected attributes in an object.