Descriptor Tutorial

Raymond Hettinger

What does the dot do in Python?

What is the difference between A.x and A.__dict__['x']?

Glyph of Warding
Abjuration

This powerful inscription harms those who enter, pass, or open the warded area or object.

Glyphs cannot be affected or bypassed by such means as physical or magical probing, though they can be dispelled.

Spell Glyph:
You can store any harmful spell of 3rd level or lower that you know

Blast Glyph:
A blast glyph deals 1d8 points of damage per two caster levels

Descriptors: the magic behind Python

- Slots
- Bound and unbound methods
- Class methods and Static Methods
- Super
- Property

Understanding Descriptors is a key to understanding the language!

What is a Descriptor?

- It is like a magic glyph.
- Reading the glyph, invokes its spell.

```
v = klass.__dict__[k]
if isinstance(v, descriptor):
    invoke(v)
else:
    return v
```

Property() is a descriptor

```
class Demo(object):

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add_parts(self):
        return self.a + self.b

    total = property(add_parts)
```

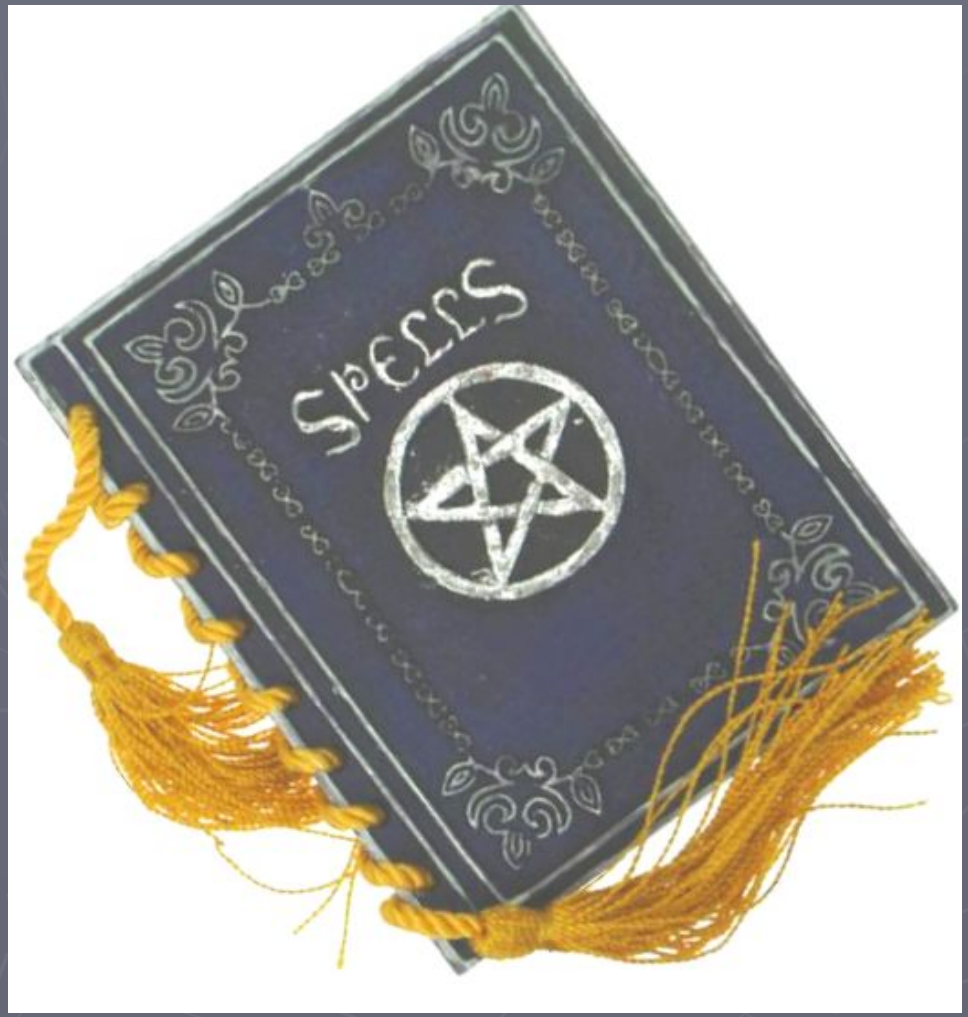
```
>>> d = Demo(10, 20)
>>> d.a
10
>>> d.b
20
>>> d.total
30
```

Magic Glyph

```
>>> vars(d)
{'a': 10, 'b': 20}

>>> Demo.__dict__['total']
<property object at 0x011AD180>
```

How do I cast my own spells?



http://tinyurl.com/d63d

The Descriptor Tutorial has ALL the gory details and complete instructions.

Technically, what is a Descriptor?

A descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol.

Those methods are __get__, __set__, and __delete__.

If any of those methods are defined for an object, it is said to be a descriptor.

Tutorial! Let's write a descriptor.

```
class Desc(object):

    def __get__(self, obj, objtype):
        print 'Invocation!'
        print 'Returning x+10'
        return obj.x+10


class A(object):

    def __init__(self, x):
        self.x = x

    plus_ten = Desc()
```

```
>>> a = A(5)

>>> a.x
5

>>> a.plus_ten
Invocation!
Returning x+10
15
```

Access from the Class

```
class Desc(object):

    def __get__(self, obj, objtype):
        print 'Invocation!'
        print 'Returning x+10'
        return obj.x+10


class A(object):

    def __init__(self, x):
        self.x = x

    plus_ten = Desc()
```

```
>>> a = A(5)

>>> A.__dict__['plus_ten']
<__main__.Desc object at
  0x011A1FF0>


>>> A.plus_ten
Invocation!
Returning x+10

Traceback (most recent call last):
  File "C:/pydev/temp_descr.py", line
    19, in __get__
    return obj.x+10
AttributeError: 'NoneType' object has
    no attribute 'x'
```

Attach it to an instance variable

```
class Desc(object):

    def __get__(self, obj, objtype):
        print 'Invocation!'
        print 'Returning x+10'
        return obj.x+10
```

```
>>> b = B(5)
>>> b.x
5
>>> b.plus_ten
<__main__.Desc object at
   0x011A1FF0>
```

```
class B(object):

    def __init__(self, x):
        self.x = x
        self.plus_ten = Desc()
```

Doesn't work with instance dicts!

Learning Points

1. Descriptor is an object defining __get__, __set__, and/or __delete__.
2. Only invoked by dotted attribute access: A.x or a.x
3. Must be stored in the class dict, not the instance dict
4. Not invoked by dictionary access: A.__dict__['x']
5. Different calling invocation for A.x and a.x

How Property is Implemented

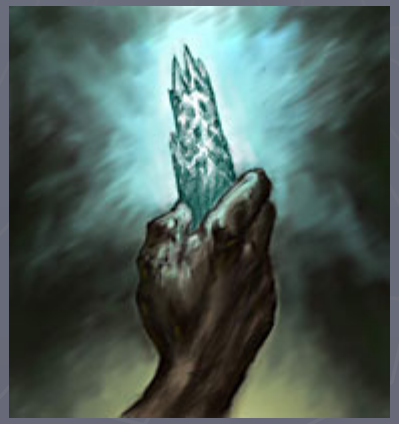
```
class Property(object):

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("object is unwriteable")
       return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)
```



Gee whiz! That's simple!

Now, you have unlimited magic

- Now you know *exactly* how property() works.
- It's just a descriptor.
- A descriptor is just an object with __get__, __set__, or __delete__.
- You could have written it yourself.
- It's trivially easy to write your own variants.

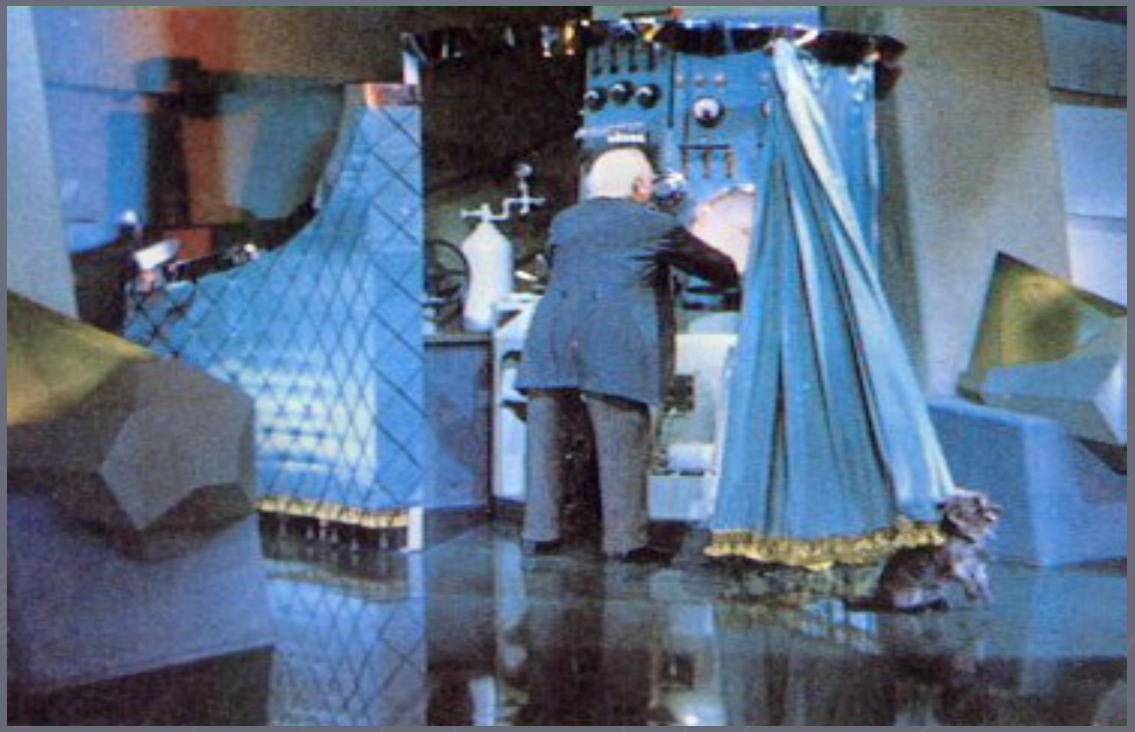
But is it *really* magic?

Why is it that A.x or a.x invokes the descriptor but A.__dict__['x'] just returns the descriptor without executing it?

How is it that A.x gets invoked differently than a.x?

Who is behind this? How do they do it?

The wizard behind the curtain



Dotted access is different from a dict lookup

A.x translates to type.__getattribute__(A, 'x')

a.x translates to object.__getattribute__(a, 'x')

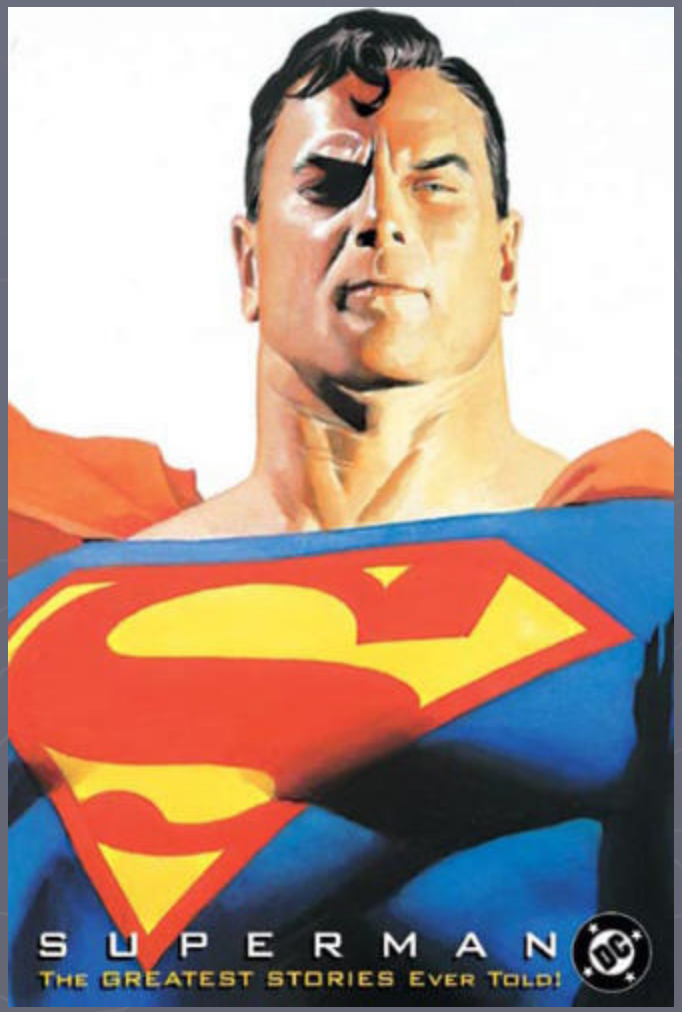
So, now the magician is revealed

- A descriptor is just an object with __get__, __set__, or __delete__.
- It is invoked by type.__getattribute__ or object.__getattribute__.
- Those methods do a dict lookup. They check if the result is a descriptor. If so, they invoke it. Otherwise, they just return the looked-up value.
- Override __getattribute__ and you can create your own new types of magic for dotted access. You own the dot.

Super!

Provides its own
__getattribute__

Its special trick is to
search the __mro__
during dotted access.



Functions!

Everybody knows how to invoke a function:

```
f(x)              # calls f with x as an argument
```

But wait! Running `dir(f)` shows that functions have a `__get__` method.

Functions are descriptors!

If you put a function in a class dictionary, the `__get__` method will activate upon dotted access.

That's how functions turn into unbound methods, bound methods, class methods and static methods!

Slots!

Worked-out example: http://tinyurl.com/59e2gk

When a class is created, the `type` metaclass assigns descriptors for each slot.

When an instance is created, space is pre-allocated for each slot.

Upon dotted access, a.x, the descriptor is invoked and fetches the value from the pre-allocated slot.

That's all there is to it.

```
class Member(object):
    'Descriptor implementing slot lookup'
    def __init__(self, i):
        self.i = i
    def __get__(self, obj, type=None):
        return obj._slotvalues[self.i]
    def __set__(self, obj, value):
        obj._slotvalues[self.i] = value
```

How Python Works

The `type` metaclass controls how classes are created. It supplies them with `__getattribute__`.

Dotted attribute access like `A.x` or `a.x` calls the `__getattribute__` method.

The `__getattribute__` method does a dict lookup, and either returns the result or invokes it if it is a descriptor (any object implementing __get__, __set__, or __delete__).

Everything else is derived from these three precepts: property, super, bound and unbound methods, class methods and static methods, and slots.

What you've learned

How to write your own descriptors

How to override __getattribute__

How all the major language features work

How to create your own magic

Not bad for 45 minutes ☺

Glyph of Warding

Creating some new General Purpose Decscriptors

- Common Property
- Cached Property
- Observerable Attributes

Redundant Properties

```
class Employee(object):
    def __init__(self, name, dept, salary, age, gender):
        self.name=name; self.dept=dept
        self._salary=salary; self._age = age; self._gender=gender

    @property
    def age(self):
        if self.permission:
            return self._age
        raise SecurityError(403, 'Access denied')

    @property
    def gender(self):
        if self.permission:
            return self._gender
        raise SecurityError(403, 'Access denied')
```

Common Property

```
class Employee(object):
    def __init__(self, name, dept, salary, age, gender):
        self.name=name; self.dept=dept
        self._salary=salary; self._age = age; self._gender=gender

    def check_permission(self, name):
        if self.permission:
            return getattr(self, name)
        raise SecurityError(403, 'Access denied')

    salary = CommonProperty('_salary', check_permission)
    age = CommonProperty('_age', check_permission)
    gender = CommonProperty('_gender', check_permission)
```

Implementation of CommonProperty()

```
class CommonProperty(object):
    """A more flexible version of property()"""

    def __init__(self, realname, fget=getattr):
        self.realname = realname
        self.fget = fget

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj, self.realname)
```

Use Case for Cached Property

```
class Bond(object):

    @CachedProperty
    def opening_price(self):
        price, = self.cursor.execute(OpenQuery,
                                     self.symbol)
        return price
```

Implementation of CachedProperty()

```
class CachedProperty(object):
    "Underlying function is called once, then cached"
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        cache_name = '_cache_' + self.func.__name__
        try:
            return getattr(obj, cache_name)
        except AttributeError:
            x = self.func(obj)
            setattr(obj, cache_name, x)
            return x
```

Observable Attributes

```
class Rectangle(object):
    length = Observable('length')
    width = Observable('width')
    def __init__(self):
        self.length = self.width = 0


>>> r = Rectangle()
>>> def track_length_updates(x, hist=[]):
...     hist.append(x)
...     print 'Length history:', hist
>>> subscribers(r, 'length').append(track_length_updates)
>>> r.length = 20
Length history: [20]
>>> r.width = 50
>>> r.length = 30
Length history: [20, 30]
```

Implementation of Observable

```
class Observable(object):
    def __set__(self, obj, value):
        if obj is None:
            return self

        oldvalue = getattr(obj, self.realname, object())
        if value != oldvalue:
            setattr(obj, self.realname, value)

            subscr_name = self.realname + '_subscribers'
            subscribers = getattr(obj, subscr_name, [])
            for callback in subscribers:
                callback(value)
```

Helper Function

```
def subscribers(obj, name):
    "Return an object's subscribers.  Create one if not found"

    subscr_name = '_' + name + '_subscribers'
    try:
        subscribers = getattr(obj, subscr_name)
    except AttributeError:
        subscribers = []
        setattr(obj, subscr_name, subscribers)
    return subscribers
```

Observer Pattern
Full Featured Recipe

http://code.activestate.com/recipes/576979

Includes thread locking and weak references to callback lists so that they don't prevent an object from getting deallocated.