Managed Attributes

Why Manage Attributes?

- Object attributes are central to most Python programs - they are where we often store information about the entities our scripts process.
- Normally, attributes are simply names for objects; a person's name attribute, for example, might be a simple string, fetched and set with basic attribute syntax.

```
person.name                 # Fetch attribute value
person.name = value         # Change attribute value
```

```
class Person:
    def getName(self):
        if not valid():
            raise TypeError('cannot fetch name')
        else:
            return self.name.transform()

    def setName(self, value):
        if not valid(value):
            raise TypeError('cannot change name')
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')
```

Inserting Code to Run on Attribute Access

- A better solution would allow you to run code automatically on attribute access, if needed.
- That's one of the main roles of managed attributes - they provide ways to add attribute accessor logic after the fact.
- More generally, they support arbitrary attribute usage modes that go beyond simple data storage.

The four accessors

- The __getattr__ and __setattr__ methods, for routing undefined attribute fetches and all attribute assignments to generic handler methods.
- The __getattribute__ method, for routing all attribute fetches to a generic handler method.
- The property built-in, for routing specific attribute access to get and set handler functions.
- The descriptor protocol, for routing specific attribute accesses to instances of classes with arbitrary get and set handler methods, and the basis for other tools such as properties and slots.

The four accessors

- All four techniques share goals to some degree, and it's usually possible to code a given problem using any one of them.
- They do differ in some important ways, though.
- For example, the last two techniques listed here apply to specific attributes, whereas the first two are generic enough to be used by delegation-based proxy classes that must route arbitrary attributes to wrapped objects.

Properties

- The property protocol allows us to route a specific attribute's get, set, and delete operations to functions or methods we provide, enabling us to insert code to be run automatically on attribute access, intercept attribute deletions, and provide documentation for the attributes if desired.
- Properties are created with the property built-in and are assigned to class attributes, just like method functions. Accordingly, they are inherited by subclasses and instances, like any other class attributes.
- Their access-interception functions are provided with the self instance argument, which grants access to state information and class attributes available on the subject instance.

Properties

- A property manages a single, specific attribute; although it can't catch all attribute accesses generically, it allows us to control both fetch and assignment accesses and enables us to change an attribute from simple data to a computation freely, without breaking existing code.
- As we'll see, properties are strongly related to descriptors; in fact, they are essentially a restricted form of them.

The Basics

- A property is created by assigning the result of a built-in function to a class attribute:

```
attribute = property(fget, fset, fdel, doc)
```

A First Example

- To demonstrate how this translates to working code, the following class uses a property to trace access to an attribute named name; the actual stored data is named _name so it does not clash with the property.

```
class Person:                                    # Add (object) in 2.X
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('fetch...')
        return self._name
    def setName(self, value):
        print('change...')
        self._name = value
    def delName(self):
        print('remove...')
        del self._name
    name = property(getName, setName, delName, "name property docs")
```

```
bob = Person('Bob Smith')          # bob has a managed attribute
print(bob.name)                    # Runs getName
bob.name = 'Robert Smith'          # Runs setName
print(bob.name)

del bob.name                       # Runs delName

print('-'*20)
sue = Person('Sue Jones')          # sue inherits property too
print(sue.name)
print(Person.name.__doc__)         # Or help(Person.name)
```

```
c:\code> py -3 prop-person.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

Computed Attributes

```
class PropSquare:
    def __init__(self, start):
        self.value = start
    def getX(self):                          # On attr fetch
        return self.value ** 2
    def setX(self, value):                   # On attr assign
        self.value = value
    X = property(getX, setX)                 # No delete or docs

P = PropSquare(3)        # Two instances of class with property
Q = PropSquare(32)       # Each has different state information

print(P.X)               # 3 ** 2
P.X = 4
print(P.X)               # 4 ** 2
print(Q.X)               # 32 ** 2 (1024)
```

Coding Properties with Decorators

- As of Python 2.6 and 3.0, property objects also have getter, setter, and deleter methods that assign the corresponding property accessor methods and return a copy of the property itself.
- We can use these to specify components of properties by decorating normal methods too, though the getter component is usually filled in automatically by the act of creating the property itself.

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):                    # name = property(name)
        "name property docs"
        print('fetch...')
        return self._name

    @name.setter
    def name(self, value):             # name = name.setter(name)
        print('change...')
        self._name = value

    @name.deleter
    def name(self):                    # name = name.deleter(name)
        print('remove...')
        del self._name
```

```
bob = Person('Bob Smith')          # bob has a managed attribute
print(bob.name)                    # Runs name getter (name 1)
bob.name = 'Robert Smith'          # Runs name setter (name 2)
print(bob.name)
del bob.name                       # Runs name deleter (name 3)

print('-'*20)
sue = Person('Sue Jones')          # sue inherits property too
print(sue.name)
print(Person.name.__doc__)         # Or help(Person.name)
```

In fact, this code is equivalent to the first example in this section—decoration is just an alternative way to code properties in this case. When it's run, the results are the same:

```
c:\code> py -3 prop-person-deco.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

Descriptors

- Descriptors provide an alternative way to intercept attribute access; they are strongly related to the properties discussed in the prior section.
- Really, a property is a kind of descriptor - technically speaking, the property built-in is just a simplified way to create a specific type of descriptor that runs method functions on attribute accesses.
- In fact, descriptors are the underlying implementation mechanism for a variety of class tools, including both properties and slots.

Descriptors

- Descriptors are created as independent classes, and they are assigned to class attributes just like method functions.
- Like any other class attribute, they are inherited by subclasses and instances.
- Their access-interception methods are provided with both a self for the descriptor instance itself, as well as the instance of the client class whose attribute references the descriptor object.
- Because of this, they can retain and use state information of their own, as well as state information of the subject instance. For example, a descriptor may call methods available in the client class, as well as descriptor-specific methods it defines.

Descriptors

- Like a property, a descriptor manages a single, specific attribute; although it can't catch all attribute accesses generically, it provides control over both fetch and assignment accesses and allows us to change an attribute name freely from simple data to a computation without breaking existing code.
- Unlike properties, descriptors are broader in scope, and provide a more general tool.
- For instance, because they are coded as normal classes, descriptors have their own state, may participate in descriptor inheritance hierarchies, can use composition to aggregate objects, and provide a natural structure for coding internal methods and attribute documentation strings.

The Basics

- As mentioned previously, descriptors are coded as separate classes and provide specially named accessor methods for the attribute access operations they wish to intercept - get, set, and deletion methods in the descriptor class are automatically run when the attribute assigned to the descriptor class instance is accessed in the corresponding way.

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): ...          # Return attr value
    def __set__(self, instance, value): ...          # Return nothing (None)
    def __delete__(self, instance): ...              # Return nothing (None)
```

A First Example

- To see how this all comes together in more realistic code, let's get started with the same first example we wrote for properties.
- The following defines a descriptor that intercepts access to an attribute named name in its clients.
- Its methods use their instance argument to access state information in the subject instance, where the name string is stored.
- Like properties, descriptors work properly only for new-style classes, so be sure to derive both classes in the following from object if you're using 2.X - it's not enough to derive just the descriptor, or just its client

```
class Name:                                    # Use (object) in 2.X
    "name descriptor docs"
    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name
    def __set__(self, instance, value):
        print('change...')
        instance._name = value
    def __delete__(self, instance):
        print('remove...')
        del instance._name

class Person:                                  # Use (object) in 2.X
    def __init__(self, name):
        self._name = name
    name = Name()                              # Assign descriptor to attr
```

```
bob = Person('Bob Smith')
print(bob.name)
bob.name = 'Robert Smith'
print(bob.name)
del bob.name

print('-'*20)
sue = Person('Sue Jones')
print(sue.name)
print(Name.__doc__)
```

```
# bob has a managed attribute
# Runs Name.__get__
# Runs Name.__set__

# Runs Name.__delete__


# sue inherits descriptor too

# Or help(Name)
```

```
c:\code> py -3 desc-person.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name descriptor docs
```

```
class Person:
    def __init__(self, name):
        self._name = name

    class Name:                                    # Using a nested class
        "name descriptor docs"
        def __get__(self, instance, owner):
            print('fetch...')
            return instance._name
        def __set__(self, instance, value):
            print('change...')
            instance._name = value
        def __delete__(self, instance):
            print('remove...')
            del instance._name
    name = Name()
```

How Properties and Descriptors Relate

- As mentioned earlier, properties and descriptors are strongly related - the property built-in is just a convenient way to create a descriptor.

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel                                    # Save unbound methods
        self.__doc__ = doc                                  # or other callables

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")
        return self.fget(instance)                          # Pass instance to self
                                                            # in property accessors
    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(instance, value)

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

```
class Person:
    def getName(self): print('getName...')
    def setName(self, value): print('setName...')
    name = Property(getName, setName)                    # Use like property()

x = Person()
x.name
x.name = 'Bob'
del x.name
```

```
c:\code> py -3 prop-desc-equiv.py
getName...
setName...
AttributeError: can't delete attribute
```

__getattr__ and __getattribute__

- __getattr__ is run for undefined attributes—because it is run only for attributes not stored on an instance or inherited from one of its classes, its use is straightforward.
- __getattribute__ is run for every attribute—because it is all-inclusive, you must be cautious when using this method to avoid recursive loops by passing attribute accesses to a superclass.
- The __getattr__ and __getattribute__ methods are also more generic than properties and descriptors—they can be used to intercept access to any (or even all) instance attribute fetches, not just a single specific name. B

The Basics

- If a class defines or inherits the following methods, they will be run automatically when an instance is used in the context described by the comments to the right:

```
def __getattr__(self, name):          # On undefined attribute fetch [obj.name]
def __getattribute__(self, name):     # On all attribute fetch [obj.name]
def __setattr__(self, name, value): # On all attribute assignment [obj.name=value]
def __delattr__(self, name):          # On all attribute deletion [del obj.name]
```

A First Example

- Generic attribute management is not nearly as complicated as the prior section may have implied.
- To see how to put these ideas to work, here is the same first example we used for properties and descriptors in action again, this time implemented with attribute operator overloading methods.
- Because these methods are so generic, we test attribute names here to know when a managed attribute is being accessed; others are allowed to pass normally.

```
class Person:                                  # Portable: 2.X or 3.X
    def __init__(self, name):                  # On [Person()]
        self._name = name                      # Triggers __setattr__!

    def __getattr__(self, attr):               # On [obj.undefined]
        print('get: ' + attr)
        if attr == 'name':                     # Intercept name: not stored
            return self._name                  # Does not loop: real attr
        else:                                  # Others are errors
            raise AttributeError(attr)

    def __setattr__(self, attr, value):        # On [obj.any = value]
        print('set: ' + attr)
        if attr == 'name':
            attr = '_name'                     # Set internal name
        self.__dict__[attr] = value            # Avoid looping here

    def __delattr__(self, attr):               # On [del obj.any]
        print('del: ' + attr)
        if attr == 'name':
            attr = '_name'                     # Avoid looping here too
        del self.__dict__[attr]                # but much less common
```

```
bob = Person('Bob Smith')          # bob has a managed attribute
print(bob.name)                    # Runs __getattr__
bob.name = 'Robert Smith'          # Runs __setattr__
print(bob.name)
del bob.name                       # Runs __delattr__

print('-'*20)
sue = Person('Sue Jones')          # sue inherits property too
print(sue.name)
#print(Person.name.__doc__)        # No equivalent here
```

```
c:\code> py -3 getattr-person.py
set: _name
get: name
Bob Smith
set: name
get: name
Robert Smith
del: name
--------------------
set: _name
get: name
Sue Jones
```

Example: Attribute Validations

p.1256 (Learning Python 5th Edition)