ECE326

PROGRAMMING LANGUAGES

Lecture 9 : Managed Attributes in Python

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Managed Attributes

- Provides control over attribute access
  - i.e. fetch (get), assignment (set), or deletion (del)
- Property
  - Turns attribute access into method invocation
  - Makes calling method appear like attribute access
    - E.g. a.foo() *vs.* a.foo
  - Created using `property` built-in function

Property

```
class Person:
    def __init__(self, first, last):
        self.first, self.last = first, last

    def get_full(self):
        return self.first + " " + self.last

    def set_full(self, value):
        self.first, self.last = value.split(" ", maxsplit=1)

    def del_full(self):
        del self.first
        del self.last

    full_name = property(get_full, set_full, del_full)
```

Property

```
class Person:
    … other definitions …
    full_name = property(get_full, set_full, del_full)

>> p = Person(“John”, “Doe”)
>> p.full_name
‘John Doe’
>> del p.full_name
>> p.first
AttributeError: ‘Person' object has no attribute 'first'
>> p.full_name = “Jane Smith”
>> p.last
‘Smith’
>> p.first
‘Jane’
```

Property

- Advice
  - Don’t use if the method performs expensive computation
    - The method is called every time you access the field
    - Property “hides” the fact that it’s actually a method call
- Use cases
  - Make field read-only
    - Do not supply the setter and deleter
  - Interface change
    - E.g. the field was accessed directly. Now you want to change the way it is used or accessed

Property

- Example: throw an error if Celsius below absolute zero

```
class Temperature:
    def __init__(self, value):
        self.celsius = value     # this will call the property

    def get_celsius(self):
        return self._celsius

    def get_fahrenheit(self):
        return self._celsius * 9 / 5 + 32

    def set_celsius(self, value):
        if value < -273.15:
            raise ValueError(“Cannot go below 273.15 degrees C”)
        self._celsius = value

    celsius = property(get_celsius, set_celsius)
```

Descriptor

- A class that customizes get, set, and/or delete of another object's attribute
- Similar to property, except more flexible
  - Since it's a class, it can be subclassed, or inherit another

```
class Descriptor:
    def __get__(self, instance, owner): …
    def __set__(self, instance, value): …
    def __delete__(self, instance): …

class Foo:
    managed = Descriptor()

f = Foo()
f.managed = 5     # calls Descriptor.__set__
```

Descriptor

- `__get__(self, instance, owner)`
  - *Instance*: the object that accesses the descriptor, `None` if attribute is accessed through the class (e.g. f.attr vs. Foo.attr)
  - *owner* is always the class (e.g. Foo)

```
>> f.managed
# self: Descriptor instance, instance: f, owner: Foo
>> Foo.managed
# self: Descriptor instance, instance: None, owner: Foo
```

- `__set__(self, instance, value)`
  - If not defined, allows descriptor to be shadowed!
    - Reminder: name resolves to instance variable before class variable

Descriptor

```
class CreditCard:
    NUM_DIGITS = 16
    def __init__(self, name, number):
        self.name, self.number = name, number
    class Number:
        def __get__(self, inst, owner):
            return inst._number[:-4] + '*****'
        def __set__(self, inst, value):
            value = value.replace('-', '')
            if len(value) != inst.NUM_DIGITS:
                raise ValueError('invalid credit card number')
            inst._number = value
    number = Number()

card = CreditCard("Jack", "1234-3453-5256-1758")
print(card.number) # prints 123434535256****
```

__setattr__

- Intercepts all assignments to the object's attribute
- Example

```
class Immutable:
    def __setattr__(self, name, value):
        raise AttributeError("cannot update read-only object")

    def __init__(self, x, y):
        self.__dict__['x'], self.__dict__['y'] = x, y


>> obj = Immutable(5, 6)
>> obj.x = 3
AttributeError: cannot update read-only object
```

__getattr__

- Intercepts all fetch (get) from an object that results in attribute not found
  - Before the AttributeError is raised
- Use case
  - Returning default values on attribute not found
  - Automatic forwarding
- Caveat
  - Does not intercept if method overloads a magic method
    - Anything that starts and ends with __ (e.g. __getitem__)

Automatic Forwarding

```
class Hand:
    def __init__(self, cards=tuple()):
        self.cards = list(cards)          # copy the list

    def _points(self):
        return sum(self.cards)
    points = property(_points)

    def __getattr__(self, name):
        return getattr(self.cards, name)

>> p = Hand([2, 3, 4])
>> p.append(9)                    # goes through __getattr__
>> print(p.points)                # points exists – does not go
18                                # through __getattr__
```

__getattribute__

- Intercepts all fetch (get) from an object
  - Also includes those not found (i.e. __getattr__)
- Danger – improper use will result in infinite recursion
  - Use super() instead of self to avoid infinite recursion
- Similar caveat as __getattr__
  - May be bypassed by operator overloading
- Use case
  - Disable access to “private” members

Private Members

```
class Protected:
    def __init__(self, x, y):
        self._x, self._y = x, y

    def getX(self):
        return vars(self)['_x']    # same as self.__dict__[‘_x’]

    def __getattribute__(self, name):
        val = super().__getattribute__(name)
        if name != "__dict__" and name.startswith("_"):
            raise AttributeError(name + “ is a private member")
        return val
```

```
>> p = Protected(5, 7)
>> p._x
AttributeError: _x is a private member
```

```
>> p.getX()
5
```