Idiomatic Python

Properties and Descriptors

Richard E Sarkis
Rochester's Python User Group
September 17th, 2013 Meeting

Overview

- Python has many powerful built-in capabilities
  - Comprehensions
  - Properties
  - Decorators
  - *…many more*

Overview

- In particular, many are familiar with *decorators* and *properties*
- …but not descriptors, a mysterious feature of the core language.
  - Esoteric, and not-well-known use cases
  - Odd syntax
  - Hard to find examples in OSS world

Descript(or)(ion)

In general, 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.

The Point

- Descriptors are reusable properties, defining a protocol for object access
- Allows you to call custom methods when trying to access, assign to or delete an instance
- This is an awesome thing…but why?

Figure 1

Properties

Properties

- Properties masquerade function calls as attributes
- Let's say we want to organize information about movies, and we construct a class as seen in Figure 2.
- In the use of our class, we'd like to prevent the assignment of negative numbers to movies. How do we forbid this?

Figure 2

Properties Use Case

- What if other parts of our code assign to `Movie.budget` directly?
  - Our class catches negative numbers on `__init__` only.
  - But not when someone tries assign

```
m.budget = -100
```

Figure 3

Properties Solve The Problem!

- Introducing: *setters* and *getters* (and *deleters*)
- *getter* methods are specified with the `@property` decorator
- *setter* methods, with a `@budget.setter` decorator
- *deleter* methods, with a `@budget.deleter` decorator

Figure 4

Properties Solve The Problem!

- Without this, we'd have:
  - To hide our instance attributes (but Python doesn't have private namespaces!)
  - Create explicit `set_budget` and `get_budget` methods

Figure 5

Nice, eh?

Or not. Don't go bananas yet!

- Properties are not reusable
- If we wanted to add a positive number check to `rating`, `runtime` and `gross` fields, we'd have this…

Figure 6

Descriptors

Enter Descriptors

- Descriptors solve the reusability problem of properties
    - Generalizes your property logic into separate classes
    - For example…

Figure 7

New Syntax

- The `NonNegative` class is a descriptor, because it defines `__get__`, `__set_` or `__delete__`
- Look how nice the `Movie` class looks now!
    - *and* negative values are detected across the board

Accessing a descriptor

- 'Getting' with descriptors
  - If we `print m.budget` Python recognizes the descriptor through its `__get__` method.
    - So, instead of passing `m.budget` directly to `print`, `Movie.budget.__get__` is called instead, passing to `print` the return value of that call.
- It is similar to how properties work — indirect method calling

Accessing a descriptor

- The `__get__` method takes two arguments (using `m.budget` as our example):
  - The instance object on the left — `m`
  - The class object of that instance — `Movie`
    - Called the 'owner' of the descriptor
    - also: `Movie.budget.__get__(None, Movie)`

Figure 8

Assigning to a descriptor

- 'Setting' with descriptors
  - If we set `m.rating = 100` Python recognizes the descriptor through its `__set__` method.
  - Won't overwrite the descriptor object assigned to `m.rating`
  - Instead, `Movie.rating.__set__(m, 100)` is called.

Assigning to a descriptor

- The `__set__` method takes two arguments (using `m.rating = 100` as our example):
  - The instance object on the left — `m`
  - The value assigned — `100`

Figure 9

Deleting with a descriptor

- 'Deleting' with descriptors
  - If we set `del(m.runtime)` Python recognizes the descriptor through its `__delete__` method.
  - Won't delete the descriptor object assigned to `m.runtime`
    - Instead, `Movie.runtime.__delete__(m)` is called.

Putting it together

- Each instance of `NonNegative` maintains a `WeakKeyDictionary` to map owner instances to data values

- When `m.budget` is called, the `__get__` method looks up data associated with m and returns the result

- The `__set__` method is similar, but has the non-negative check

Putting it together

- Why use `WeakKeyDictionary`?
    - May cause a memory leak of holding a reference to an object sitting unused in the descriptors dictionary

Putting it together

- Descriptors are required to be assigned at the class-level
  - Because of this, every instance of our class `Movie` will share the same instance of the descriptors.
  - That's why we pass in the object reference when calling `__get__`, `__set__`, and `__delete__`

Semi-Conclusion

- Properties and descriptors are powerful tools for idiomatic Python programming
- If you find that your properties are repeating the same logic, try refactoring to descriptors

Tips

- **Descriptors at the class level**
  - They must be defined at the class level, otherwise `__get__`, `__set__`, and `__delete__` won't be invoked

Figure 10

Tips

- **Descriptors need to handle multiple instances**
  - Each instance of the class using descriptors needs to store and reference instance-specific values assigned to it.
  - Hence, the dictionary we discussed
  - This is the most awkward bit of descriptors

Tips

- **Beware un-hashable descriptor owners**
  - The MoProblems class is subclassed from list, which isn't a hashable object
    - As such, they cannot be used as keys in a dictionary

Figure 12

Tips

- **Beware un-hashable descriptor owners**
  - We can get around this with 'labeling' our descriptors
    - Without descriptors, Python would access `f.x` as `f.__dict__['x']`
    - With descriptors, this is not used so we can safely store our values in that key
- It is fragile, subtle and apparently common.

Figure 14

Tips

- **Labeled descriptors with Metaclasses**
  - Since descriptor labels match the variable name they are assigned to, metaclasses can handle the bookkeeping automatically.
  - A bit beyond the scope of our talk

Figure 15

Tips

- **Accessing descriptor methods**
  - Descriptors are just classes
  - However, `__get__`, `__set__`, and `__delete__` are always called, shrouding any access to other methods, and thus unreachable!
  - The solution is to attack this from the class-level

Figure 16

Demystification

- When looking up a member using x.y, Python searches for the member in the instance dictionary
  - Failing that, it looks for it in the class dictionary
  - If it is in the class dictionary, and implements the descriptor protocol, it goes for it

Figure 16

Conclusion

- Descriptors are used in Python to implement properties, bound methods, static methods, class methods and slots, and more
- They're used everywhere!

Figure 17

References

- Python Descriptors Demystified
  - http://bit.ly/RocPySept17Ref1
- Descriptor HowTo Guide
  - http://bit.ly/RocPySept17Ref2