Discovering Descriptors

Mariano Anaya

EuroPython - July 2017

rmariano

rmarianoa

def

“Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design”.

- Raymond Hettinger

Introduction

In general:

```
>>> obj = DomainModel()
>>> obj.x = 'value'
>>> obj.x
'value'
```

Control Access to Data

But what if…

When doing “`obj.x`” we could run arbitrary code?

Control Access to Data

But what if…

When doing “`obj.x`” we could run arbitrary code?

By another object.

Control Access to Data

But what if…

When doing “`obj.x`” we could run arbitrary code?

By another object (of a different class).

A First Look at Descriptors

Introduction

Descriptors enable control over core operations (get, set, delete), of an attribute in an object.

Descriptor Methods

```
__get__(self, instance, owner)
__set__(self, instance, value)
__delete__(self, instance)
__set_name__(self, owner, name) *
```

* Python 3.6

David Beazley
@dabeaz

Following

Also, the first run of the advanced course on Python 3.6. There's so much cool new stuff with descriptors/class decorators to explore.

RETWEETS	LIKES
2	14

2:39 PM - 2 Apr 2017

2 14

Types of Descriptors

- **Non-data** descriptors (a.k.a “non-overriding”)
  - Don’t implement `__set__`
  - Instance attributes take precedence
- **Data** descriptors (a.k.a. “overriding”)
  - Implement `__get__`, `__set__`
  - Override instance’s `__dict__`

`__get__`

Problem: automatically format date values of other attributes.

Two classes:

Descriptor + Managed class

Descriptor

```python
class DateFormatter:
    FORMAT = "%Y-%m-%d %H:%M"

    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        if instance is None:
            return self
        date_value = getattr(instance, self.name)
        if date_value is None:
            return ''
        return date_value.strftime(self.FORMAT)
```

Managed Class

```
class FileStat:
    """Stats of a file in a virtual file system"""
    str_created_at = DateFormatter('created_at')
    str_updated_at = DateFormatter('updated_at')
    str_removed_at = DateFormatter()

    def __init__(self, fname, created, updated=None,
removed=None):
        self.filename = fname
        self.created_at = created
        self.updated_at = updated
        self.removed_at = removed
```

```
>>> created = updated = datetime(2017, 6, 9, 11, 15, 19)
>>> f1 = FileStat('/home/mariano/file1', created, updated)

>>> f1.str_created_at
'2017-06-09 11:15'
>>> f1.str_updated_at
'2017-06-09 11:15'
>>> f1.str_removed_at
''
```

Resolution Order

Statement

```
>>> f1 = FileStat(...)
>>> f1.str_created_at
```

```
f1.__dict__
{
    'created_at': ...
    'filename': '/home/...',
    'removed_at': ...,
    'updated_at': ...
}
```

Statement

```
>>> f1 = FileStat(...)
>>> f1.str_created_at
```

```
FileStat.__dict__
mappingproxy({'__dict__': ...,
              '__doc__': "...",
              '__init__': ...,
              'str_created_at': <DateFormatter at 0x..>,
              'str_removed_at': <DateFormatter at 0x..>,
              'str_updated_at': <DateFormatter at 0x..>})
```

Statement

```
>>> f1 = FileStat(...)
>>> f1.str_created_at
```

```
>>> hasattr(FileStat.__dict__['str_created_at'], '__get__')
True
```

`__get__` : Syntax Sugar

```
>>> f1 = FileStat(...)
>>> f1.str_created_at
```

Translates into:

```
FileStat.str_created_at.__get__(f1, FileStat)
```

Access Through the Class

```
__get__(self, instance, owner)
```

When called like `<class>.<descriptor>`

`instance` **`is None`**

```
>>> FileStat.str_created_at
<__main__.DateFormatter object at 0x...>
```

Name of the Descriptor

```
class FileStat:
    """Stats of a file in a virtual file system"""
    str_created_at = DateFormatter('created_at')
    str_updated_at = DateFormatter('updated_at')
    str_removed_at = DateFormatter()
```

Before `__set_name__`

Some techniques to have an “automatic configuration”:

Class decorator or metaclass

`__set_name__(self, owner, name)`

Called automatically with the name of the attribute, on the LHS.

```
class owner:
    name = Descriptor()
```

__set_name__

```
class DateFormatter:
    def __init__(self, name=None):
        self.name = name
    ...
    def __set_name__(self, owner, name):
        if self.name is None:
            _, _, self.name = name.partition('_')
```

`__set__`

Problem: Given an attribute of an object, keep count of how many times its value was changed.

Data Descriptor: `__set__`

Some strategies:

1. Properties (with setter)
2. Override `__setattr__()`
3. **`Descriptors`**!

```
class TracedProperty:
    """Count how many times an attribute changed its value"""
    def __set_name__(self, owner, name):
        self.name = name
        self.count_name = f'count_{name}'

    def __set__(self, instance, value):
        ...
```

```
class TracedProperty:
    ...
    def __set__(self, instance, value):
        try:
            current_value = instance.__dict__[self.name]
        except KeyError:
            instance.__dict__[self.count_name] = 0
        else:
            if current_value != value:
                instance.__dict__[self.count_name] += 1

        instance.__dict__[self.name] = value
```

```
class Traveller:
    city = TracedProperty()
    country = TracedProperty()

    def __init__(self, name):
        self.name = name
```

```
>>> tourist = Traveller('John Smith')
>>> tourist.city = 'Barcelona'
>>> tourist.country = 'Spain'

>>> tourist.count_city
0
>>> tourist.count_country
0
```

```
>>> tourist.city = 'Stockholm'
>>> tourist.country = 'Sweden'
>>> tourist.count_city
1
>>> tourist.count_country
1
```

```
>>> tourist.city = 'Gothenburg'
>>> tourist.count_city
2
>>> tourist.count_country
1
>>> tourist.country = 'Sweden'
>>> tourist.count_country
1
```

`__set__`: Syntax sugar

```
tourist = Traveller()
tourist.city = 'Stockholm'
```

Translates to:

```
Traveller.city.__set__(tourist, 'Stockholm')
```

`__delete__`

Called when deleting an attribute by using the descriptor, like:

```
del <instance>.<descriptor>
```

```python
class ProtectedAttribute:
    """Attribute that is protected against deletion"""

    def __set_name__(self, owner, name):
        self.name = name

    def __delete__(self, instance):
        raise AttributeError(f"Can't delete {self.name} for
{instance!s}")

    def __set__(self, instance, value):
    ...
```

```
class ProtectedUser:
    username = ProtectedAttribute()

    def __init__(self, username, location):
        self.username = username
        self.location = location

    def __str__(self):
        return f"{self.__class__.__name__}[{self.username}]"
```

```
>>> usr = ProtectedUser('jsmith', '127.0.0.1')
>>> usr.username
'jsmith'
>>> del usr.username
Traceback (most recent call last):
...
AttributeError: Can't delete username for ProtectedUser[jsmith]
>>> usr.location
'127.0.0.1'
>>> del usr.location
>>> usr.location
Traceback (most recent call last):
...
AttributeError: 'ProtectedUser' object has no attribute 'location'
```

What makes a good descriptor?

What makes a good descriptor?

The same thing that makes any good Python object: consistency with Python itself (to be *Pythonic*).

Descriptors in CPython

Descriptors are deployed in the language infrastructure.

- `@property`, `@classmethod`, `@staticmethod`
- `Methods (functions)`

Functions are Descriptors

They have a `__get__` method.

That's why they can work as instance methods!

`<function>.__get__` returns the function bound to an object.

```
class Class:
    def method(self, *args):
        return f'{self!s} got {args}'


>>> Class.__dict__
mappingproxy({'__dict__': ...
              'method': <function Class.method>})


>>> isinstance(Class.__dict__['method'], types.FunctionType)
True
```

Method Call

```
>>> instance = Class()
>>> instance.method('arg1', 'arg2')
"instance got ('arg1', 'arg2')"
```

It’s actually...

```
>>> Class.method.__get__(instance, Class)('arg1', 'arg2')
"instance got ('arg1', 'arg2')"
```

Extended Uses

Improve decorators that change the signature.

Apply to Functions & Methods as well

Problem: A decorator that changes the signature, has to work both for functions and methods.

E.g. abstract away repeated code.

```python
def resolver_function(root, args, context, info):

    helper = DomainObject(root, args, context, info)
    ...
    helper.process()
    helper.task1()
    helper.task2()
    return helper.task1()
```

```python
class DomainArgs:
    def __init__(self, func):
        self.func = func
        wraps(func)(self)

    def __call__(self, root, args, context, info):
        helper = DomainObject(root, args, context, info)
        return self.func(helper)

@DomainArgs
def resolver_function(helper):
    helper.task1()
    ...
```

Try to Decorate a Method

```
class ViewResolver:
    @DomainArgs
    def resolve_method(self, helper):
        response = helper.process()
        return f"Method: {response}"
```

```
>>> vr1.resolve_method('root', 'args', 'context', 'info')
-----------------------------------
TypeError   Traceback (most recent call last)

     39     def __call__(self, root, args, context, info):
     40         helper = DomainObject(root, args, context, info)
---> 41         return self.func(helper)
     42

TypeError: resolve_method() missing 1 required positional argument:
'helper'
```

Doesn’t handle **self**!

Fix: __get__

```
class DomainArgs:
    ...
    def __get__(self, instance, owner):
        mapped = self.func.__get__(instance, owner)
        return self.__class__(mapped)

>>> vr = ViewResolver()
>>> vr.method_resolver('root', 'args', 'context', 'info')
'Method resolver: root, args, context, info'
```

Closing Remarks

Implement the minimum required interface.

Use for general-purpose solutions.

Thanks!

@rmarianoa