effective python 29-32

박진성

목차

```
class OldResistor(object):
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```

파이썬 답지가 않다!

```
r0 = OldResistor(3)
print('Before: %5r' % r0.get_ohms())
r0.set_ohms(10)
print('After:  %5r' % r0.get_ohms())
```

```
Before:     3
After:     10
```

```
# 아래와 같이 사용하기엔 불편하다.
r0.set_ohms(r0.get_ohms() + 10)
```

```python
# 아래와 같이 속성을 사용하면 편하다.
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0
```

```python
r1 = Resistor(3)
print('Before: %5r' % r1.ohms)
r1.ohms = 20
print('After:  %5r' % r1.ohms)
```

```
Before:     3
After:     20
```

```python
# 속성을 이용하여 += 연산도 쉽게 사용
r1.ohms += 20
```

```python
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 3

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

```python
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0
```

```python
r2 = VoltageResistance(3)
print('Before: %5r amps' % r2.current)
r2.voltage = 10
print('After: %5r amps' % r2.current)
```

```
Before:     0 amps
After: 3.3333333333333335 amps
```

```python
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError('%f ohms must be > 0' % ohms)
        self._ohms = ohms
```

```python
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0
```

```python
r3 = BoundedResistance(3)
r3.ohms = 0
```

```
 File "C:\Users\jinsung\PycharmProjects\EffectivePython\Chapter29\Chapter29.py", line 85, in <module>
    r3.ohms = 0
  File "C:\Users\jinsung\PycharmProjects\EffectivePython\Chapter29\Chapter29.py", line 81, in ohms
    raise ValueError('%f ohms must be > 0' % ohms)
ValueError: 0.000000 ohms must be > 0
```

```python
class FixedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Can't set attribute")
        self._ohms = ohms
```

```python
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0
```

```python
r4 = FixedResistance(3)
r4.ohms = 3
```

```
 File "C:\Users\jinsung\PycharmProjects\EffectivePython\Chapter29\Chapter29.py", line 107, in <module>
    r4.ohms = 3
  File "C:\Users\jinsung\PycharmProjects\EffectivePython\Chapter29\Chapter29.py", line 103, in ohms
    raise AttributeError("Can't set attribute")
AttributeError: Can't set attribute
```

```
class MysteriousResistor(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        self._ohms = ohms
```

```
class Resistor(object):
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0
```

```
r7 = MysteriousResistor(20)
r7.current = 0.01
print('Before: %5r' % r7.voltage)
r7.ohms
print('After:  %5r' % r7.voltage)
```

```
Before:     0
After:    0.2
```

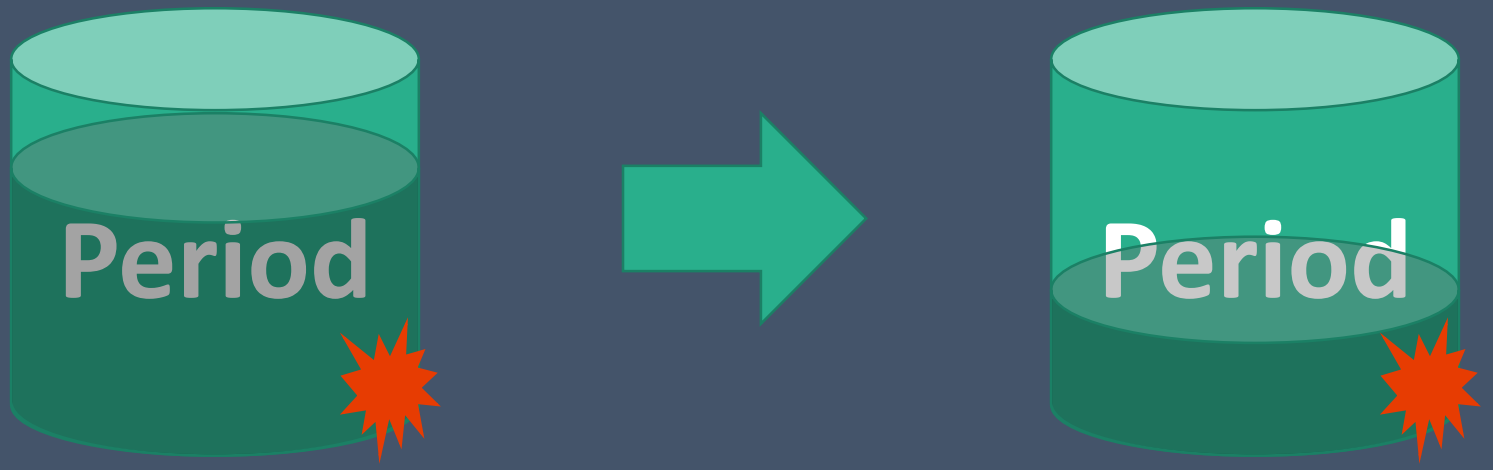
결론

- 게터와 세터보단 간단한 공개 속성을 사용하자
- 속성에 접근할 때 @property를 통해 특별한 동작이 가능하다
- @property는 최대한 간단한 작업만 수행하자

목차

```
class Bucket(object):
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return 'Bucket(quota=%d)' % self.quota
```



```
def fill(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

```
def deduct(bucket, amount):
    now = datetime.now()
    if now - bucket.reset_time > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)
```

```
Bucket(quota=100)
```


```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print(bucket)
```

```
Had 99 quota
Bucket(quota=1)
```

```
if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)
```

```
Not enough for 3 quota
Bucket(quota=1)
```

```python
@property
def quota(self):
    return self.max_quota - self.quota_consumed

@quota.setter
def quota(self, amount):
    delta = self.max_quota - amount
    if amount == 0:
        # 새 기간의 할당량을 리셋함
        self.quota_consumed = 0
        self.max_quota = 0
    elif delta < 0:
        # 새 기간의 할당량을 채움
        assert self.quota_consumed == 0
        self.max_quota = amount
    else:
        # 기간 동안 할당량을 소비함
        assert self.max_quota >= self.quota_consumed
        self.quota_consumed += delta
```

```
bucket = Bucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)
```

```
Initial Bucket(max_quota=0, quota_consumed=0)
Filled Bucket(max_quota=100, quota_consumed=0)
```

```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print('Now', bucket)
```

```
Had 99 quota
Now Bucket(max_quota=100, quota_consumed=99)
```

```
if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print('Still', bucket)
```

```
Not enough for 3 quota
Still Bucket(max_quota=100, quota_consumed=99)
```

결론

- 속성에 새 기능을 넣을 때 @property를 고려하자
- @property 메서드가 점점 커진다면 리팩토링을 고려하자

목차

```python
class Homework(object):
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._grade = value
```

```
class Exam(object):
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
```

```
@property
def writing_grade(self):
    return self._writing_grade

@writing_grade.setter
def writing_grade(self, value):
    self._check_grade(value)
    self._writing_grade = value
```

```
@property
def math_grade(self):
    return self._math_grade

@math_grade.setter
def math_grade(self, value):
    self._check_grade(value)
    self._math_grade = 0
```

디스크립터 프로토콜 (descriptor protocol)

- 속성에 대한 접근을 언어에서 해석할 방법을 정의한다.

```
class Grade(object):
    def __init__(self):
        self._value = 0

    def __get__(self, instance, owner):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._value = value
```

```
class Exam(object):
    writing_grade = Grade()
    math_grade = Grade()
    science_grade = Grade()
```

```
exam = Exam()
exam.writing_grade = 40
print(exam.writing_grade)
```

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
Exam.__dict__['writing_grade'].__get__(exam, Exam)
```

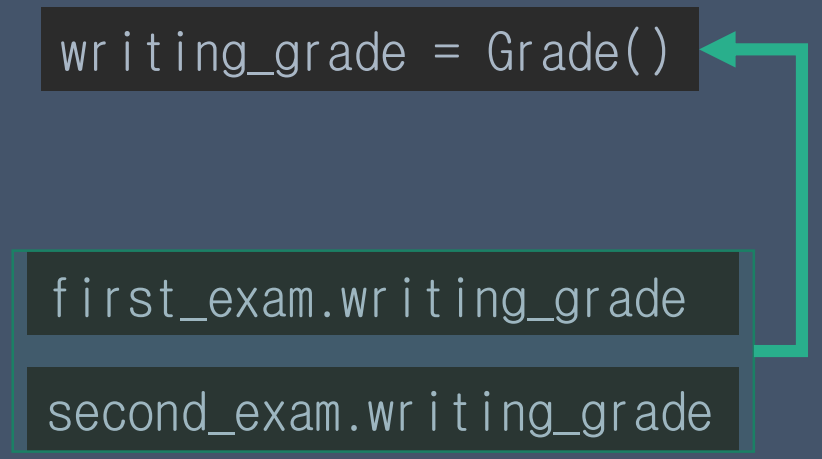
```
class Exam(object):
    writing_grade = Grade()
    math_grade = Grade()
    science_grade = Grade()
```

```
first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)
```

```
Writing 82
Science 99
```

```
second_exam = Exam()
second_exam.writing_grade = 75
print('Second', second_exam.writing_grade, 'is right')
print('First', first_exam.writing_grade, 'is wrong')
```

```
Second 75 is right
First 75 is wrong
```



```python
class Grade(object):
    def __init__(self):
        self._values = {}

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self._values.get(instance, 0)


    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._values[instance] = value
```

Grade로 인한 instance의 참조 계수 1유지..

GC가 메모리 수집을 못한다!!

Grade에 있는 instance들은 우주 속으로..

weakref

1. 파이썬 내장 모듈
2. WeakKeyDictionary : 런타임에 마지막 남은 Exam 인스턴스의 참조를 갖고 있다는 사실을 알면 키 집합에서 Exam 인스턴스를 제거
    - 파이썬이 대신 참조를 관리

```
from weakref import WeakKeyDictionary
```

```
class Grade(object):
    def __init__(self):
        self._values = WeakKeyDictionary()

    # ...
```

```
first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print('First ', first_exam.writing_grade, 'is right')
print('Second ', second_exam.writing_grade, 'is right')
```

```
First  82 is right
Second  75 is right
```

결론

- 디스크립터를 이용하여 @property 메서드를 재활용하자
- WeakKeyDictionary를 사용하여 메모리 누수가 없도록하자

목차

```
class LazyDB(object):
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        value = 'Value for %s' % name
        setattr(self, name, value)
        return value
```

```
data = LazyDB()
print('Before:', data.__dict__)
print('foo:   ', data.foo)
print('After: ', data.foo)
```

```
Before: {'exists': 5}
```

```
foo:    Value for foo
```

```
After:  Value for foo
```

```
class LoggingLazyDB(LazyDB):
    def __getattr__(self, name):
        print('Called __getattr__(%s)' % name)
        return super().__getattr__(name)
```

```
data = LoggingLazyDB()
print('exists:', data.exists)
print('foo:   ', data.foo)
print('foo:   ', data.foo)
```

```
exists: 5
```

```
Called __getattr__(foo)
```

```
foo:    Value for foo
```

```
foo:    Value for foo
```

```
class ValidatingDB(object):
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        try:
            return super().__getattribute__(name)
        except AttributeError:
            value = 'Value for %s' % name
            setattr(self, name, value)
            return value
```

```
data = ValidatingDB()
print('exists:', data.exists)
print('foo:   ', data.foo)
print('foo:   ', data.foo)
```

```
Called __getattribute__(exists)
exists: 5
Called __getattribute__(foo)
foo:    Value for foo
Called __getattribute__(foo)
foo:    Value for foo
```

```
class MissingPropertyDB(object):
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError('%s is missing' % name)
        # ...
```

```
class LoggingLazyDB(LazyDB):
    def __getattr__(self, name):
        print('Called __getattr__(%s)' % name)
        return super().__getattr__(name)
```

```
data = LoggingLazyDB()
print('Before:      ', data.__dict__)
print('foo exists   ', hasattr(data, 'foo'))
print('After:       ', data.__dict__)
print('foo exists   ', hasattr(data, 'foo'))
```

```
Before:      {'exists': 5}
```

```
Called __getattr__(foo)
```

```
foo exists   True
```

```
After:       {'exists': 5, 'foo': 'Value for foo'}
```

```
foo exists   True
```

```
class SavingDB(object):
    def __setattr__(self, name, value):
        # 몇몇 데이터를 DB 로그로 저장함
        super().__setattr__(name, value)
```

```
class LoggingSavingDB(SavingDB):
    def __setattr__(self, name, value):
        print('Called __setattr__(%s, %r)' % (name, value))
        super().__setattr__(name, value)
```

```
data = LoggingSavingDB()
print('Before:  ', data.__dict__)
data.foo = 5
print('After:   ', data.__dict__)
data.foo = 7
print('Finally  ', data.__dict__)
```

```
Before:   {}
Called __setattr__(foo, 5)
After:    {'foo': 5}
Called __setattr__(foo, 7)
Finally   {'foo': 7}
```

```python
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = {}

    def __getattribute__(self, name):
        print('Called __getattribute__(%s) %', name)
        return self._data[name]
```

```python
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        data_dict = super().__getattribute__('_data')
        return data_dict[name]
```

결론

- 객체의 속성을 지연 방식으로 로드할 땐 __getattr__과 __setattr__을 사용하자.
- __getattribute__와 __setattr__에서 super() 를 사용하여 무한 재귀 호출을 막자.

끝