파이썬 스터디

9주차

Ch33. 메타클래스로 서브클래스를 검증하자

Ch34. 메타클래스로 클래스의 존재를 등록하자

Ch35. 메타클래스로 클래스 속성에 주석을 달자

메타클래스 동작

- type을 상속하여 정의
- 타입이 실제로 생성되기 전에 클래스 정보를 수정 가능
- __new__ 메서드에서 연관된 class문의 정보를 받음
- 클래스의 이름, 클래스가상속하는 부모 클래스, class 본문에서 정의한 모든 클래스 속성에 접근 가능

- http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python
- https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

```python
import logging
from pprint import pprint
from sys import stdout as STDOUT


# Example 1
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        orig_print = __builtins__.print
        print = pprint
        print((meta, name, bases, class_dict))
        print = orig_print
        return type.__new__(meta, name, bases, class_dict)

class MyClass(object, metaclass=Meta):
    stuff = 123

    def foo(self):
        pass
```

```
(<class '__main__.Meta'>,
 'MyClass',
 (<class 'object'>,),
 {'__module__': '__main__',
  '__qualname__': 'MyClass',
  'foo': <function MyClass.foo at 0x008E7108>,
  'stuff': 123})
```

서브 클래스 검증

- 메타 클래스는 서브 클래스가 정의될 때마다 검증할 수 있음
- 클래스를 올바르게 정의했는지 검증
- 클래스 계층을 만들 때 스타일을 강제
- 메서드를 오버라이드 하도록 요구
- 클래스 속성 사이에 철저한 관계를 둠
- 오류를 더 빨리 일으킴(인스턴스 생성이 아닌 선언 단계에서)

```python
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Don't validate the abstract Polygon class
        if bases != (object,):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(object, metaclass=ValidatePolygon):
    sides = None  # Specified by subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3
```

```python
print(Triangle.interior_angles())
try:
    print('Before class')
    class Line(Polygon):
        print('Before sides')
        sides = 1
        print('After sides')
    print('After class')
except:
    logging.exception('Expected')
else:
    assert False
```

```
180
Before class
Before sides
After sides
ERROR:root:Expected
ValueError: Polygons need 3+ sides
```

결론

- 메타클래스의 __new__ 메서드는 class 문의 본문 전체가 처리된 후에 실행됨
- 서브클래스 타입의 객체를 생성하기에 앞서 서브클래스가 정의 시점부터 제대로 구성되었음을 보장하려면 메타클래스를 사용하자
- 파이썬2와 파이썬3의 메타클래스 문법이 다름

```
class MyClassInPython2(object):
    __metaclass__ = Meta
    stuff = 123

    def foo(self):
        pass
```

Ch33. 메타클래스로 서브클래스를 검증하자

Ch34. 메타클래스로 클래스의 존재를 등록하자

Ch35. 메타클래스로 클래스 속성에 주석을 달자

타입 자동 등록

- 프로그램에 있는 타입을 자동으로 등록할 때 메타클래스를 사용할 수 있음
- 객체들의 매핑이 필요한 경우 메타클래스가 유용함

```python
import json

class Serializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})

class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point2D(%d, %d)' % (self.x, self.y)

point = Point2D(5, 3)
print('Object:     ', point)
print('Serialized:', point.serialize())
```

```
Object:     Point2D(5, 3)
Serialized: {"args": [5, 3]}
[Finished in 0.6s]
```

```python
class Serializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})

class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])

class BetterPoint2D(Deserializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return 'BetterPoint2D(%d, %d)' % (self.x, self.y)
```

```python
point = BetterPoint2D(5, 3)
print('Before:    ', point)
data = point.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After:     ', after)
```

```
Before:     BetterPoint2D(5, 3)
Serialized: {"args": [5, 3]}
After:      BetterPoint2D(5, 3)
[Finished in 0.6s]
```

문제점

- Serializable된 데이터에 대응하는 타입(위의 경우 Point2D, BetterPoint2D)을 미리 알고 있을 때만 동작함 (?!)

해결 방안

- 어떤 클래스든 대응할 수 있는 Deserializable 공통함수로 변경하고 직렬화할 객체의 클래스 이름을 JSON 데이터에 포함시킴

```python
class BetterSerializable(object):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
        return '%s(%s)' % (
            self.__class__.__name__,
            ', '.join(str(x) for x in self.args))

registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

```python
register_class(BetterPoint2D)
print('registry:  ', registry)
point = BetterPoint2D(5, 3)
print('Before:    ', point)
data = point.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After:     ', after)
```

```
registry:   {'BetterPoint2D':
    <class '__main__.BetterPoint2D'>}
Before:     BetterPoint2D(5, 3)
Serialized: {"args": [5, 3],
             "class": "BetterPoint2D"}
After:      BetterPoint2D(5, 3)
```

문제점

- 어떤 클래스를 담고 있는지 몰라도 임의의 JSON 문자열을 Deserialize 할 수 있음
- Deserialize가 항상 제대로 동작함을 보장하려면, 추후에 역질렬화할 법한 모든 클래스에 register_class를 호출해야함
- register_class 호출을 잊어먹을 수 있음

```python
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(object, metaclass=Meta):
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
        return '%s(%s)' % (
            self.__class__.__name__,
            ', '.join(str(x) for x in self.args))

class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z
```

```python
v3 = Vector3D(10, -7, 3)
print('Before:    ', v3)
data = v3.serialize()
print('Serialized:', data)
print('After:     ', deserialize(data))
```

```
Before:     Vector3D(10, -7, 3)
Serialized: {"args": [10, -7, 3],
             "class": "Vector3D"}
After:      Vector3D(10, -7, 3)
```

결론

- 메타클래스를 이용하면, 서브클래스를 만들 때마다 자동으로 등록 코드를 실행 가능
- 클래스 등록은 모듈 방식을 만들 때 유용
- 메타클래스를 이용해 클래스 등록하면, 오류 방지 가능

Ch33. 메타클래스로 서브클래스를 검증하자

Ch34. 메타클래스로 클래스의 존재를 등록하자

Ch35. 메타클래스로 클래스 속성에 주석을 달자

개요

- 메타클래스를 사용하여 클래스 생성 후, 클래스 사용 전에 프로퍼티를 수정하거나 주석을 붙일 수 있다.
- 보통 디스크립터(__get__, __set__)와 함께 사용한다.

```python
class Field(object):
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)

class Customer(object):
    # Class attributes
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

```python
foo = Customer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euclid'
print('After: ', repr(foo.first_name), foo.__dict__)
```

```
Before: '' {}
After:  'Euclid' {'_first_name': 'Euclid'}
```

개선 필요

- Field 디스크립터가 인스턴스 딕셔너리 __dict__를 수정함
- Customer 클래스 내부에서 필드를 생성할 때 필드의 이름을 명시했는데, 생성자에도 적어야 함

```
first_name = Field('first_name')
```

```python
class Field(object):
    def __init__(self):
        # 메타클래스가 이 속성들을 할당함
        self.name = None
        self.internal_name = None
    def __get__(self, instance, instance_type):
        if instance is None: return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)

class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls

class DatabaseRow(object, metaclass=Meta):
    pass

class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

```python
foo = BetterCustomer()
print('Before:', repr(foo.first_name), foo.__dict__)
foo.first_name = 'Euler'
print('After: ', repr(foo.first_name), foo.__dict__)
```

```
Before: '' {}
After:  'Euler' {'_first_name': 'Euler'}
```

결론

- 메타클래스를 이용하여 클래스가 완전히 정의되기 전에 클래스 속성을 수정할 수 있다
- 디스크립터와 메타클래스는 선언적 동작과 런타임 내부 조사(introspection)용으로 강력한 조합을 이룬다.