6. Metaclasses and Attributes

Simply put, metaclasses let you intercept Python’s class statement and provide special behavior each time a class is defined.

(Metaclasses는 파이썬의 클래스 선언을 가로채고, 클래스가 선언될 때마다 특별한 행동을 제공한다.)

▼ Item 44: Use Plain Attributes Instead of Setter and Getter Methods

Getter and Setter

> 🔑 AI 프로그래밍 할 때 본격적으로 파이썬에서 클래스를 사용했었는데 그때 썩 효율적이지 않게 코딩했다는 것을 알게 되었다.

파이썬에서는 명시적인 getter나 setter를 선언할 필요 없다.

```
# 잘못된 예시
class OldResistor:
    def __init__(self, ohms):
        self._ohms = ohms

    def getOhms(self):
        return self._ohms

    def setOhms(self, ohms):
        self._ohms = ohms

r0 = OldResistor(50e3)
print('Before: ', r0.getOhms())
r0.setOhms(10e3)
print('After: ', r0.getOhms())
```

> 🅰️ Before: 50000.0
> After: 10000.0

implementations들을 public attributes로 설정하라.

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

r1 = Resistor(50e3)
r1.ohms = 10e3
r1.ohms += 5e3
print('수정 후: ', r1.ohms)
```

```
수정 후: 15000.0
```

Later, if I decide I need special behavior when an attribute is set, I can migrate to the @property decorator and its corresponding setter attribute.

```
# @property 사용
class VoltageResistacne(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

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

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

r2 = VoltageResistacne(1e3)
print(f'Before: {r2.current:.2f} amps')
r2.voltage = 10
print(f'After: {r2.current:.2f} amps')
```

```
Before: 0.00 amps
After: 0.01 amps
```

setter를 구체화 하면 타입 체크가 가능해진다.

생성자를 잘못 생성해도 에러가 발생한다. __init__ 이 Resistor.__init__을 호출하는데, 그러면 self.ohms에 마이너스 값이 할당되니까 @ohms.setter 메소드가 에러를 발생시킨다.

```
# Specifying a setter
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; got {ohms}')
        self._ohms = ohms

r3 = BoundedResistance(1e3)
r3.ohms = 0
BoundedResistance(-5)
```

```
Traceback (most recent call last):
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 68, in <module>
r3.ohms = 0
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 64, in ohms
raise ValueError(f'ohms must be > 0; got {ohms}')
ValueError: ohms must be > 0; got 0
```

🅰️ Traceback (most recent call last):
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 69, in <module>
BoundedResistance(-5)
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 55, in **init**
super().**init**(ohms)
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 21, in **init**
self.ohms = ohms
File "d:\OneDrive - pusan.ac.kr\PNU\2-2.5\AI Study\Effective Python\06_Metaclasses_Attributes\item 44.py", line 64, in ohms
raise ValueError(f'ohms must be > 0; got {ohms}')
ValueError: ohms must be > 0; got -5

@property 사용해서 부모 클래스의 특성을 수정하지 못하게 할 수 있다.

- hasattr()
  - 변수가 있는지 확인하는 함수
  - 있으면 True, 없으면 False 반환
  - self에 '_ohms'라는 변수가 있는지 확인하고, 있으면 에러를 발생시킨다.

```
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("Ohms is immutable")
        self._ohms = ohms

r4 = FixedResistance(1e3)
r4.ohms = 2e3
```

When you use @property methods to implement setters and getters, be sure that the behavior you implement is not surprising.

```
# behavior you implement is not surprising
class MysteriousResistor(Resistor):
    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms

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

r7 = MysteriousResistor(10)
r7.current = 0.01
print(f'Before: {r7.voltage:.2f}')
r7.ohms
print(f'After: {r7.voltage:.2f}')
```

🅰️ Before: 0.00
After: 0.10

[Things to Remember]

- Define new class interfaces using simple public attributes and avoid defining setter and getter methods.
- Use @property to define special behavior when attributes are accessed on your objects, if necessary.
- Follow the rule of least surprise and avoid odd side effects in your @property methods.
- Ensure that @property methods are fast; for slow or complex work—especially involving I/O or causing side effects—use normal methods instead.

▼ Item 45: Consider @property Instead of Refactoring Attributes

@property는 좀 더 똑똑하게 행동하게 하는 인스턴스의 특성에 간단히 접근할 수 있게 한다.

@property 사용은 단순한 수치 속성이었던 것을 즉각적인 계산으로 전환할 수 있게 한다. 이것은 아주 유용한데, 그것은 모든 존재하는 클래스의 사용을 다시 쓰여진 call sites를 요청하지 않고 새로운 작동을 이동시킬 수 있기 때문이다.

For example, say that I want to implement a leaky bucket quota using plain Python objects. Here, the Bucket class represents how much quota remains and the duration for which the quota will be available:

```
# implement a leaky bucket quota
from datetime import datetime, timedelta

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

    def __repr__(self):
        return f'Bucket(quota={self.quota})'
```

leaky bucket 알고리즘은 버킷이 가득 찼을 때, 할당량(quota)이 한 기간을 넘어 다음으로 전달되지 않는다는 것을 전제로 한다.

```
# whenever the bucket is filled, the amount of quota doesn't carry over
# from one period to the next
    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
```

할당량 소비자가 무언가를 하고 싶을 때마다 그것이 사용할 필요가 있는 할당량을 감하는 것을 먼저 보장해야 한다.

```
def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False # Bucket hasn't been filled this period
    if bucket.quota - amount < 0:
        return False # Bucket was filled, but not enough
    bucket.quota -= amount
    return True

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

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

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

>> Bucket(quota=100)

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

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

여기까지 하면 bucket이 어떤 quota level에서 시작하는 지 절대 알 수 없다. quota는 0이 될 때까지 계속 감소한다. 그러면 deduct()는 bucket이 다시 채워질 때까지 항상 False를 반환하게 된다.

이 문제를 해결해보자.

```
# solving quota problem
class NewBucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return (f'NewBucket(max_quota={self.max_quota}',\
            f'quota_consumed={self.quota_consumed}')
```

원래 Bucket class의 이전 인터페이스와 맞추기 위해 @property 메서드를 사용한다, 즉석에서 quota의 현재 level을 계산하기 위해서.

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

quota attribute가 할당되면 특별한 행동을 취한다, 클래스의 현재 사용에 호환되기 위해, fill()과 deduct() 함수를 사용해서.

```
    @quota.setter
    def quota(self, amount):
        delta = self.max_quota - amount
        if amount == 0:
            # Quota being reset for a new period
            self.quota_consumed = 0
            self.max_quota = 0
        elif delta < 0:
            # Quota being filled for the new period
            assert self.quota_consumed == 0
            self.max_quota = amount
        else:
            # Quota being consumed during the period
            assert self.max_quota >= self.quota_consumed
            self.quota_consumed += delta

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

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

print("Now", bucket)

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

print("Still", bucket)
```

>> Initial NewBucket(max_quota=0,quota_consumed=0)
>> Filled NewBucket(max_quota=100,quota_consumed=0)
>> Had 99 quota
>> Now NewBucket(max_quota=100,quota_consumed=99)
>> Not enough for 3 quota
>> Still NewBucket(max_quota=100,quota_consumed=99)

[Things to Remember]

- Use @property to give existing instance attributes new functionality.

- Make incremental progress toward better data models by using @property.
- Consider refactoring a class and all call sites when you find yourself using @property too heavily.

▼ Item 46: Use Descriptors for Reusable @property Methods

@property의 큰 문제점은 재사용이다. 그것이 꾸미는 메서드는 같은 클래스에서 multiple attributes로 재사용될 수 없다. 또한 연관없는 클래스에서도 재사용될 수 없다.

숙제 점수 클래스 생성

```
class Homework:
    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

galileo = Homework()
galileo.grade = 95
```

과목 별 점수 다르게

```
class Exam:
    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 = value
```

파이썬에서는 descriptor를 사용하는 게 좋다.

descriptor는 __get__과 __set__ 메서드를 제공하고, 상용구 없이 재사용할 수 있다.

item 41에서 본 mix-in 보다도 좋다. 단일 클래스에서 많은 다양한 특성들을 위해 같은 로직을 재사용할 수 있게 하기 때문이다.

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

    def __get__(self, instance, instance_type):
        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:
    math_grade = Grade()
    writing_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

다중 Exam 인스턴스의 이러한 attributes에 접근하는 것은 예상치 못한 결과를 초래한다.

```
second_exam = Exam()
second_exam.writing_grade = 75
print(f'Second {second_exam.writing_grade} is right')
print(f'First {first_exam.writing_grade} is wrong; should be 82')
```

>> Second 75 is right

>> First 75 is wrong; should be 82

- 문제: 이 attribute에 대한 Grade 인스턴스가 한 번 프로그램의 생명 주기에 들어가면 Exam 클래스가 처음으로 정의될 때 Exam 인스턴스가 만들어진다.
- 해결: Grade 클래스가 필요하다 각각의 독특한 Exam 인스턴스의 값을 추적하기 위해서.

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

    def __get__(self, instance, instance_type):
        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
```

---

leak memory 문제 해결을 위해 weakref 라는 내장 모듈을 사용한다.

```
from weakref import WeakKeyDictionary

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

    def __get__(self, instance, instance_type):
        pass

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

[Things to Remember]

- Reuse the behavior and validation of @property methods by defining your own descriptor classes.
- Use WeakKeyDictionary to ensure that your descriptor classes don’t cause memory leaks.
- Don’t get bogged down trying to understand exactly how __getattribute__ uses the descriptor protocol for getting and setting attributes.

▼ Item 47: Use __getattr__, __getattribute__, and __setattr__ for Lazy Attributes

Python’s object hooks make it easy to write generic code for gluing systems together.

(파이썬의 object hooks은 시스템을 한 데 잇기 위해 제네릭 코드를 짜는 것을 쉽게 하게 해준다.)

Python makes this dynamic behavior possible with the __getattr__ special method.

(파이썬은 __getattr__라는 특별한 메서드로 dynamic behavior를 가능하게 한다.)

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

    def __getattr__(self, name):
        value = f'Value for {name}'
        setattr(self, name, value)
        return value
```

---

missing property foo에 접근하면, __dict__ 인스턴스를 변형시키는 __getattr__를 불러오게 한다.

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

>> Before: {'exists': 5}
>> foo: Value for foo
>> After: {'exists': 5, 'foo': 'Value for foo'}

Here, I add logging to LazyRecord to show when __getattr__ is actually called.

```
class LoggingLazyRecord(LazyRecord):
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r}), '
            f'populating instance dictionary')
        result = super().__getattr__(name)
        print(f'* Returning {result!r}')
        return result

data = LoggingLazyRecord()
print('exists:      ', data.exists)
print('First foo:   ', data.foo)
print('Second foo:  ', data.foo)
```

>> exists: 5

>> Called __getattr__('foo'), populating instance dictionary

>> Returning 'Value for foo'
>> First foo: Value for foo
>> Second foo: Value for foo

{name!r}에서 !r이 뭘까? 저거 빼고 출력하면 작은 따옴표가 안 나오던데

파이썬은 다른 오브젝트인 __getattribute__라는 hook을 제공한다.

This special method is called every time an attribute is accessed on an object, even in cases where it does exist in the attribute dictionary.

This enables me to do things like check global transaction state on every property access.

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

    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        try:
            value = super().__getattribute__(name)
            print(f'* Found {name!r}, returning {value!r}')
            return value
```

```
        except AttributeError:
            value = f'Value for {name}'
            print(f'* Setting {name!r} to {value!r}')
            setattr(self, name, value)
            return value

data = ValidatingRecord()
print('exists:      ', data.exists)
print('First foo:   ', data.foo)
print('Second foo:  ', data.foo)
```

>> Called __**getattribute**__('exists')

>> Found 'exists', returning 5
>> exists: 5

>> Called __**getattribute**__('foo')

>> Setting 'foo' to 'Value for foo'
>> First foo: Value for foo

>> Called __**getattribute**__('foo')

>> Found 'foo', returning 'Value for foo'
>> Second foo: Value for foo

무한 호출을 막으려면 super()를 사용하면 된다.

```
class DictionaryRecord:
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        data_dict = super().__getattribute__('_data')
        return data_dict[name]

data = DictionaryRecord({'foo': 3})
print('foo: ', data.foo)
```

>> Called __getattribute__('foo')
>> foo: 3

[Things to Remember]

- Use __getattr__ and __setattr__ to lazily load and save attributes for an object.

- Understand that __getattr__ only gets called when accessing a missing attribute, whereas __getattribute__ gets called every time any attribute is accessed.
- Avoid infinite recursion in __getattribute__ and __setattr__ by using methods from super() (i.e., the object class) to access instance attributes.

▼ Item 48: Validate Subclasses with __init_subclass__

메타 클래스의 가장 간단한 사용 중 하나는 클래스가 올바르게 정의되었는지 검증하는 것이다.

검증을 위해 메타 클래스를 사용하는 것은 일찍이 에러를 불러낸다, 클래스를 가진 모듈이 프로그램 시작 초기에 import 됐을 때.

유효한 서브 클래스에 메타 클래스를 정의하기 전에 표준 오브젝트를 위한 메타 클래스 동작을 이해해야 한다.

메타 클래스는 type으로부터 상속되어 정의된다. default case에서 메타 클래스는 연관된 class statements의 __new__ 메서드에서 내용을 받아온다.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(f'* Running {meta}.__new__ for {name}')
        print('Bases: ', bases)
        print(class_dict)
        return type.__new__(meta, bases, class_dict)

class MyClass(metaclass=Meta):
    stuff = 123

    def foo(self):
        pass

class MySubClass(MyClass):
    other = 567

    def bar(self):
        pass
```

메타 클래스는 클래스, 부모 클래스, 그리고 클래스 body에 정의된 모든 클래스 attributes의 name에 접근한다. 모든 클래스들은 object로부터 상속된다, 그래서 base classes의 튜플에 명시적으로 적혀있지 않다.

```
>> Running <class '__main__.Meta'>.new for MyClass
>> Bases: ()
>> {'__module__': '__main__', '__qualname__': 'MyClass', 'stuff': 123, 'foo': <function MyClass.foo at 0x000001B721089360>}

>> Running <class '__main__.Meta'>.__new__ for MySubClass
>> Bases: (<class '__main__.MyClass'>,)
>> {'__module__': '__main__', '__qualname__': 'MySubClass', 'other': 567, 'bar': <function MySubClass.bar at 0x000001B7210893F0>}
```

I can add functionality to the Meta.__**new**__ method in order to validate all of the parameters of an associated class before it's defined.

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Polygon class
        if bases:
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    sides = None # Must be specified by subclasses

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

class Triangle(Polygon):
    sides = 3

class Rectangle(Polygon):
    sides = 4

class Nonagon(Polygon):
    sides = 9

assert Triangle.interior_angles() == 180
assert Rectangle.interior_angles() == 360
assert Nonagon.interior_angles() == 1260
```

If I try to define a polygon with fewer than three sides, the validation will cause the class statement to fail immediately after the class statement body.

→ 프로그램이 그런 클래스를 정의할 때는 시작조차 하지 않는다는 것을 의미한다.

```
print('Before class')
```

```
class Line(Polygon):
    print('Before sides')
    sides = 2
    print('After sides')

print('After class')
```

>> Before class
>> Before sides
>> After sides
>> Traceback…

>> ValueError: Polygons need 3+ sides

This seems like quite a lot of machinery in order to get Python to accomplish such a basic task. Luckily, Python 3.6 introduced simplified syntax—the __init_subclass__ special class method—for achieving the same behavior while avoiding metaclasses entirely.

```
class BetterPolygon:
    sides = None # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError('Polygons need 3+ sides')

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

class Hexagon(BetterPolygon):
    sides = 6

assert Hexagon.interior_angles() == 720
```

코드가 훨씬 간결해졌고 ValidatePolygon 메타 클래스는 아예 사라졌다. 또한 따라가기 쉽다, 왜냐하면 sides attribute에 직접적으로 접근할 수 있기 때문이다. __init_subclass__ 의 cls 인스턴스에 직접적으로 접근할 수 있기 때문에, class_dict['sides']를 이용해서 클래스의 딕셔너리로 가는 대신에.

```
print('Before class')

class Point(BetterPolygon):
    sides = 1

print('After class')
```

```
```

>> Before class

>> Traceback

>> ValueError: Polygons need 3+ sides

Another problem with the standard Python metaclass machinery is that you can only specify a single metaclass per class definition.

```
class ValidateFilled(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Filled class
        if bases:
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return type.__new__(meta, name, bases, class_dict)

class Filled(metaclass=ValidateFilled):
    color = None # Must be specified by subclasses
```

Polygon 메타 클래스와 Filled 메타 클래스를 함께 사용하려고 하면 아리송한(cryptic) 에러 메시지가 뜬다.

```
class RedPentagon(Filled, Polygon):
    color = 'red'
    sides = 5
```

>> Traceback

>> TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases

---

It’s possible to fix this by creating a complex hierarchy of metaclass type definitions to layer validation

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    is_root = True
```

```
    sides = None # Must be specified by subclasses

class ValidateFilledPolygon(ValidatePolygon):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return type.__new__(meta, name, bases, class_dict)

class FilledPolygon(Polygon, metaclass=ValidateFilledPolygon):
    is_root = True
    color = None # Must be specified by subclasses

class GreenPentagon(FilledPolygon):
    color = 'green'
    sides = 5

greenie = GreenPentagon()
assert isinstance(greenie, Polygon)
```

색깔에 대해 검증한다.

```
class OrangePentagon(FilledPolygon):
    color = 'orange'
    sides = 5
```

>> Traceback …

>> ValueError: Fill color must be supported

변의 개수에 대해서도 검증한다.

```
class RedLine(FilledPolygon):
    color = 'red'
    sides = 2
```

얘는 변의 개수가 2라서 3보다 작기 때문에 에러 메시지를 띄워야 하는데 안 띄워준다.

뭐가 문제지?

→ ValidatePolygon이 문제인 듯. 작동을 안 한다. 이유가 뭘까?

However, this approach ruins composability, which is often the purpose of class validation like this. If I want to apply the color validation logic from ValidateFilledPolygon to another

hierarchy of classes, I'll have to duplicate all of the logic again, which reduces code reuse and increases boilerplate.

The __init_subclass__ special class method can also be used to solve this problem. It can be defined by multiple levels of a class hierarchy as long as the super built-in function is used to call any parent or sibling __init_subclass__ definitions. It's even compatible with multiple inheritance.

```
class Filled:
    color = None

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.color not in ('red', 'green', 'blue'):
            raise ValueError('Fills need a valid color')

class RedTriangle(Filled, Polygon):
    color = 'red'
    sides = 3

ruddy = RedTriangle()
assert isinstance(ruddy, Filled)
assert isinstance(ruddy, Polygon)
```

잘못된 선언을 하면 그에 맞는 에러 메시지를 띄운다.

```
print('Before class')

class BlueLine(Filled, Polygon):
    color = 'blue'
    sides = 2

print('After class')
```

>> Before class

>> Traceback …

>> ValueError: Polygons need 3+ sides

```
print('Before class')

class BeigeSquare(Filled, Polygon):
    color = 'beige'
    sides = 4

print('After class')
```

>> Before class

>> Traceback …

>> ValueError: Fills need a valid color

You can even use __init_subclass__ in complex cases like diamond inheritance (see Item 40: “Initialize Parent Classes with super”).

```
class Top:
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Top for {cls}')

class Left(Top):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Left for {cls}')

class Right(Top):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Right for {cls}')

class Bottom(Left, Right):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Bottom for {cls}')
```

>> Top for <class '__main__.Left'>

>> Top for <class '__main__.Right'>

>> Top for <class '__main__.Bottom'>

>> Right for <class '__main__.Bottom'>

>> Left for <class '__main__.Bottom'>

[Things to Remember]

- The __new__ method of metaclasses is run after the class statement’s entire body has been processed.
- Metaclasses can be used to inspect or modify a class after it’s defined but before it’s created, but they’re often more heavyweight than what you need.
- Use __init_subclass__ to ensure that subclasses are well formed at the time they are defined, before objects of their type are constructed.

- Be sure to call super().__init_subclass__ from within your class's __init_subclass__ definition to enable validation in multiple layers of classes and multiple inheritance.

▼ Item 49: Register Class Existence with __init_subclass__

메타 클래스의 흔한 사용은 자동적으로 타입을 프로그램에 등록하는 것이다.

등록은 유용하다 반대로 검색할 때, 간단한 식별자를 대응하는 클래스에 mapping 할 때.

ex) For example, say that I want to implement my own serialized representation of a Python object using JSON. I need a way to turn an object into a JSON string. Here, I do this generically by defining a base class that records the constructor parameters and turns them into a JSON dictionary

```
import json

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

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

이 클래스는 Point2D처럼 간단하고 수정할 수 없는 데이터 구조를 연속적으로 나타내기에 쉽다.

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

    def __repr__(self) -> str:
        return f'Point2D({self.x}, {self.y})'

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

>> Object: Point2D(5, 3)

>> Serialized: {"args": [5, 3]}

---

deserialize this JSON string and construct the Point2D object it represents

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

Using Deserializable makes it easy to serialize and deserialize simple, immutable objects in a generic way

```
class BetterPoint2D(Deserializable):
    ...

before = BetterPoint2D(5, 3)
print('Before:      ', before)
data = before.serializable()
print('Serialized:  ', data)
after = BetterPoint2D.deserialize(data)
print('After:       ', after)
```

>> Before: <__main__.BetterPoint2D object at 0x00000282EB3DEE60>

>> Serialized: {"args": [5, 3]}

>> After: <__main__.BetterPoint2D object at 0x00000282EB5E44C0>

이 접근 방식의 문제점은 그것은 오직 당신이 serialized data의 의도된 타입을 미리 알고 있을 때 작동한다는 것이다. 이상적으로는 당신은 많은 클래스를 JSON으로 serializing 하고 있고, 하나의 공통된 함수가 그것들을 다시 deserialize 할 수 있어야 한다. 이것을 하기 위해 JSON data에 serialized object의 클래스 이름을 포함할 수 있다.

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

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

    def __repr__(self):
        name = self.__class__.__name__
        args_str = ', '.join(str(x) for x in self.args)
        return f'{name}({args_str})'
```

Then, I can maintain a mapping of class names back to constructors for those objects.

```
registry = {}

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

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

deserialize 가 항상 올바르게 작동함을 보장하기 위해 register_class를 나중에 deserialize 하고 싶은 모든 클래스에 불러와야 한다.

```
class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

register_class(EvenBetterPoint2D)
```

이제 임의의 JSON string을 어떤 클래스가 그것을 포함하고 있는지 알지 않아도 deserialize 할 수 있다.

```
before = EvenBetterPoint2D(5, 3)
print('Before:      ', before)
data = before.serialize()
print('Serialized:  ', data)
after = deserialize(data)
print('After:       ', after)
```

>> Before: EvenBetterPoint2D(5, 3)

>> Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}

>> After: EvenBetterPoint2D(5, 3)

이 접근 방법의 문제는 register_class의 호출을 잊어버릴 수 있다는 것이다.

```
class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z

# Forgot to call register_class!
```

This causes the code to break at runtime, when I finally try to deserialize an instance of a class I forgot to register.

```
point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)
```

>> Traceback …

>> KeyError: 'Point3D’

서브 클래스 BetterSerializable을 선택했지만 만약 class statement body 이후에 register_class을 호출하는 것을 잊어버렸다면 사실 모든 그것의 특징을 가져온 것이 아니다. 이런 접근은 에러를 발생시키는 경향이 있고 특히 초심자에게는 어렵다.

What if I could somehow act on the programmer’s intent to use BetterSerializable and ensure that register_class is called in all cases? Metaclasses enable this by intercepting the class statement when subclasses are defined.

```
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(BetterSerializable, metaclass=Meta):
    pass
```

RegisteredSerializable의 서브 클래스를 정의했을 때 register_class를 호출할 것이라고 확신할 수 있다. 그리고 deserialize는 항상 예측 범위 내에서 작동할 것이다.

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

```
        self.x, self.y, self.z = x, y, z

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

>> Before: Vector3D(10, -7, 3)

>> Serialized: {"class": "Vector3D", "args": [10, -7, 3]}

>> After: Vector3D(10, -7, 3)

__init_subclass__를 사용하면 좀 더 낫다.

```
class BetterRegisteredSerializable(BetterSerializable):
    def __init_subclass__(cls):
        super().__init_subclass__()
        register_class(cls)

class Vector1D(BetterRegisteredSerializable):
    def __init__(self, magnitude):
        super().__init__(magnitude)
        self.magnitude = magnitude

before = Vector1D(6)
print('Before:      ', before)
data = before.serialize()
print('Serialized:  ', data)
print('After:       ', deserialize(data))
```

>> Before: Vector1D(6)

>> Serialized: {"class": "Vector1D", "args": [6]}

>> After: Vector1D(6)

[Things to Remember]

- Class registration is a helpful pattern for building modular Python programs.
- Metaclasses let you run registration code automatically each time a base class is subclassed in a program.
- Using metaclasses for class registration helps you avoid errors by ensuring that you never miss a registration call.

- Prefer __init_subclass__ over standard metaclass machinery because it's clearer and easier for beginners to understand.

▼ Item 50: Annotate Class Attributes with __set_name__

메타 클래스가 가능하게 하는 유용한 것은 properties를 수정하거나 주석을 다는 것이다, 클래스가 정의된 이후이나 아직 클래스가 사용되기 전일 때. 이 접근은 흔히 descriptors와 함께 사용된다, 그들이 좀 더 많이 자기 성찰을 하게 하기 위해, 그들이 포함된 클래스 내에서 어떻게 사용되는지에 대해.

```
class Field:
    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)
```

Field descriptor에 저장된 column name과 함께 모든 per-instance state를 직접적으로 instance dictionary에 저장할 수 있다, 보호된 field에서, setattr라는 내장 함수를 사용해서, 그리고 state를 불러올 수 있다 getattr를 이용해서.

row를 나타내는 클래스를 정의하는 것은 각 클래스 attribute에 대한 데이터베이스 테이블의 column 이름을 필요로 한다.

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

Using the class is simple. Here, you can see how the Field descriptors modify the instance dictionary __dict__ as expected.

```
cust = Customer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euclid'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

>> Before: '' {}

>> After: 'Euclid' {'_first_name': 'Euclid'}

The problem is that the order of operations in the Customer class definition is the opposite of how it reads from left to right.

First, the Field constructor is called as Field('first_name').

Then, the return value of that is assigned to Customer.field_name.

There's no way for a Field instance to know upfront which class attribute it will be assigned to.
To eliminate this redundancy, I can use a metaclass.

Metaclasses let you hook the class statement directly and take action as soon as a class body is finished.

In this case, I can use the metaclass to assign Field.name and Field.internal_name on the descriptor automatically instead of manually specifying the field name multiple times.

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

메타 클래스를 사용한 base class 생성

```
class DatabaseRow(metaclass=Meta):
    pass
```

이 메타 클래스가 작동하기 위해서는 Field descriptor가 크게 바뀌지 않아야 한다.

The only difference is that it no longer requires arguments to be passed to its constructor. Instead, its attributes are set by the Meta.__new__ method above.

```
class Field:
    def __init__(self):
        # These will be assigned by the metaclass.
        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)
```

메타 클래스를 이용하면, 새 DatabaseRow base class, 새 Field descriptor, 데이터베이스 열 클래스 정의가 더 이상 이전처럼 불필요한 것을 갖지 않는다.

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

cust = BetterCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euler'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

>> Before: '' {}

>> After: 'Euler' {'_first_name': 'Euler'}

The trouble with this approach is that you can’t use the Field class for properties unless you also inherit from DatabaseRow. If you somehow forget to subclass DatabaseRow, or if you don’t want to due to other structural requirements of the class hierarchy, the code will break.

```
class BrokenCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = BrokenCustomer()
cust.first_name = 'Mersenne'
```

>> Traceback …

>> TypeError: attribute name must be string, not 'NoneType’

__set_name__ 이라는 특별한 메서드를 사용하면 해결된다.

This method is called on every descriptor instance when its containing class is defined. It receives as parameters the owning class that contains the descriptor instance and the attribute name to which the descriptor instance was assigned.

```
class FixedCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = FixedCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Mersenne'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

>> Before: '' {}

>> After: 'Mersenne' {'_first_name': 'Mersenne'}

[Things to Remember]

- Metaclasses enable you to modify a class’s attributes before the class is fully defined.
- Descriptors and metaclasses make a powerful combination for declarative behavior and runtime introspection.
- Define __set_name__ on your descriptor classes to allow them to take into account their surrounding class and its property names.
- Avoid memory leaks and the weakref built-in module by having descriptors store data they manipulate directly within a class’s instance dictionary.

▼ Item 51: Prefer Class Decorators Over Metaclasses for Composable Class Extensions

메타 클래스가 여러 방법으로 클래스를 원하는 대로 만들 수 있게 해주지만, 여전히 일어날 수 있는 모든 상황을 다루는 데는 부족하다.

ex) debugging decorator

```
from functools import wraps

def trace_func(func):
    if hasattr(func, 'tracing'): # Onlye decorate once
        return func

    @wraps(func)
    def wrapper(*args, **kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
        finally:
            print(f'{func.__name__}({args!r}, {kwargs!r}) -> '
                f'{result!r}')
    wrapper.tracing = True
    return wrapper
```

I can apply this decorator to various special methods in my new dict subclass

```
class TraceDict(dict):
    @trace_func
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @trace_func
    def __setitem__(self, *args, **kwargs):
        return super().__setitem__(*args, **kwargs)

    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)
```

이제 이 메서드들이 클래스 인스턴스와 상호작용하면서 데코레이션 되는지 검증할 수 있다.

```
trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
```

```
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

>> __init__(({'hi': 1}, [('hi', 1)]), {}) -> None

>> __setitem__(({'hi': 1, 'there': 2}, 'there', 2), {}) -> None

>> __getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1

>> __getitem__(({'hi': 1, 'there': 2}, 'does not exist'), {}) -> KeyError('does not exist')

The problem with this code is that I had to redefine all of the methods that I wanted to decorate with @trace_func.

→ One way to solve this problem is to use a metaclass to automatically decorate all methods of a class.

```
class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super().__new__(meta, name, bases, class_dict)

        for key in dir(klass):
            value = getattr(klass, key)
            if isinstance(value, trace_types):
                wrapped = trace_func(value)
                setattr(klass, key, wrapped)
        return klass
```

Now, I can declare my dict subclass by using the TraceMeta metaclass and verify that it works as expected.

```
class TraceDict(dict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

>> __new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}

This works, and it even prints out a call to __new__ that was missing from my earlier implementation. What happens if I try to use TraceMeta when a superclass already has specified a metaclass?

```
class OtherMeta(type):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass
```

>> Traceback …

>> TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases

→ TraceMeta가 OtherMeta에 상속되어 있지 않기 때문에 실패한 것이다.

In theory, I can use metaclass inheritance to solve this problem by having OtherMeta inherit from TraceMeta.

```
class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super().__new__(meta, name, bases, class_dict)

        for key in dir(klass):
            value = getattr(klass, key)
            if isinstance(value, trace_types):
                wrapped = trace_func(value)
                setattr(klass, key, wrapped)
        return klass

class OtherMeta(TraceMeta):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
```

```
except KeyError:
    pass # Expected
```

But this won’t work if the metaclass is from a library that I can’t modify, or if I want to use multiple utility metaclasses like TraceMeta at the same time.

→ To solve this problem, Python supports class decorators. Class decorators work just like function decorators: They’re applied with the @ symbol prefixing a function before the class declaration.

```
def my_class_decorator(klass):
    klass.extra_param = 'hello'
    return klass

@my_class_decorator
class MyClass:
    pass

print(MyClass)
print(MyClass.extra_param)
```

can implement a class decorator to apply trace_func to all methods and functions of a class by moving the core of the TraceMeta.__new__ method above into a stand-alone function.

→ 메타 클래스 버전보다 훨씬 짧아진다.

```
def trace(klass):
    for key in dir(klass):
        value = getattr(klass, key)
        if isinstance(value, trace_types):
            wrapped = trace_func(value)
            setattr(klass, key, wrapped)
    return klass
```

I can apply this decorator to my dict subclass to get the same behavior as I get by using the metaclass approach above.

```
@trace
class TraceDict(dict):
    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

trace_dict = TraceDict([('hi', 1)])
```

```
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

>> __new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}

>> __getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1

>> __getitem__(({'hi': 1, 'there': 2}, 'does not exist'), {}) -> KeyError('does not exist')

Class decorators also work when the class being decorated already has a metaclass.

```
class OtherMeta(type):
    pass

@trace
class TraceDict(dict, metaclass=OtherMeta):
    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

>> __new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}

>> __getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1

>> __getitem__(({'hi': 1, 'there': 2}, 'does not exist'), {}) -> KeyError('does not exist')

클래스를 확장하기 위한 편리한 방법을 찾는다면, 클래스 데코레이터가 가장 적합한 도구이다.

[Things to Remember]

- A class decorator is a simple function that receives a class instance as a parameter and returns either a new class or a modified version of the original class.
- Class decorators are useful when you want to modify every method or attribute of a class with minimal boilerplate.
- Metaclasses can’t be composed together easily, while many class decorators can be used to extend the same class without conflicts.