effective python, class!

2016.11.4

이영득

Covers with..

22. 딕셔너리와 튜플보다는 헬퍼 클래스로 관리하자.
    (Appendix: namedtuple)

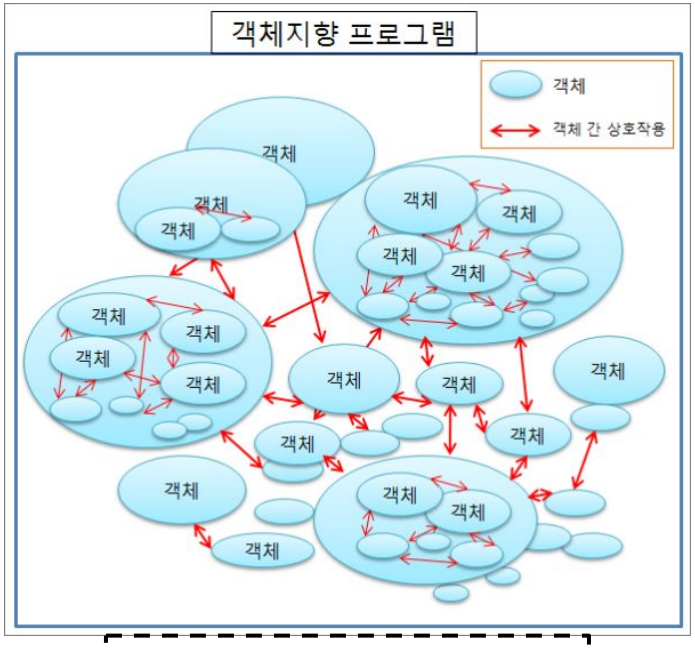
23. 인터페이스가 간단하면 클래스 대신 함수를 받자.
    (Appendix: lambda, callable)

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

25. super 로 부모 클래스를 초기화하자.

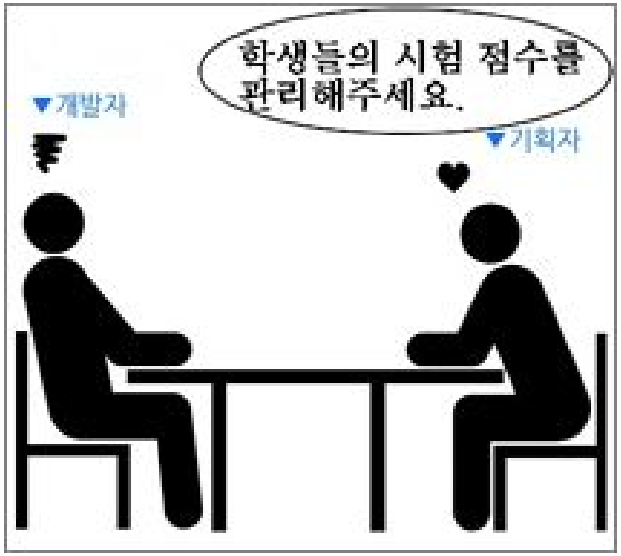
22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.

‘근데, 사실, 보다 보니까..
python 이야기라기 보다
객체 지향 모델링에 관한
이야기 인 것 같아요..’



(이런 느낌적인 느낌)

22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.



GradeBook: class (1 .. 1) _grades: dict

_grades: dict (1 .. N)

key = name: string

value = score: array<int>

```python
class SimpleGradeBook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)

book = SimpleGradeBook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)

print(book.average_grade('Isaac Newton'))
```

22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.

GradeBook: class (1 .. 1) _grades: dict

_grades: dict (1 .. N)

key = name: string

value = subject: dict

key = subject: string

value = score: array<int>

```python
class BySubjectGradeBook(object):
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = {}  # 여기 바뀜 [] => {}

    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
        total, count = 0, 0
        for grades in by_subject.values():
            total += sum(grades)
            count += len(grades)
        return total / count
```

22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.

GradeBook: class (1 .. 1) _grades: dict
_grades: dict (1 .. N)
key = name: string
value = subject: dict
key = subject: string
value = score: array
array<tuple(score, weight)>

```python
class WeightedGradeBook(object):
    # ...
    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))

    def average_grade(self, name):
        by_subject = self._grades[name]
        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
            for score, weight in scores:
                # ...
                subject_avg += score
                total_weight += weight
        return 0
```

22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.

'잠깐, 내 코드가 왜 이런 모양이 되었지?'

⇒ 각 object들을 클래스로 뽑아내는 것이 현명하다.
실제 코드는 기니까, pycharm으로 한번 보자.

22. 딕셔너리와 튜플보다는 헬퍼클래스로 관리하자.

책에 있는 정리

- 다른 딕셔너리나 긴 튜플을 값으로 담은 딕셔너리를 생성하지 말자.
- 정식 클래스의 유연성이 필요 없다면 가벼운 불변 데이터 컨테이너에는 namedtuple을 사용하자.
- 내부 상태를 관리하는 딕셔너리가 복잡해지면 여러 헬퍼 클래스를 사용하는 방식으로 관리 코드를 바꾸자.

22-1. Appendix. namedtuple?

이름과 키-벨류 조합으로 사용한다.

- 값을 중간에 바꿀 수 없다.
- 정의한 키 (name, age, type)이 외의 다른 키를 사용하거나,
  기존 키 값을 비워둘 수 없다.
- tuple과 마찬가지로 index 로 접근할 수 있다. (perry[0] ⇒ perry.name)
- 사용법은 dict 와 비슷하다. _asdict() 함수로 변환해서 사용할 수 있다.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')

perry = Animal(name="perry", age=31, type="cat")

# Animal = namedtuple('Animal', 'name age')  # Error!
# Animal = namedtuple('Animal', 'name age type new_variable')  # Error!
print(perry)  # Output: Animal(name='perry', age=31, type='cat')
print(perry.name)  # Output: 'perry'
# perry.name = 'gordonlee'  # Error!

print(perry[0])  # fine

print(perry._asdict())
```

23. 인터페이스가 간단하면 클래스 대신 함수를 받자.

```
def log_missing():
    print('key added')
    return 0

current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]
result = defaultdict(log_missing, current)
print('Before: {0}'.format(dict(result)))
for key, amount in increments:
    result[key] += amount
print('After: {0}'.format(dict(result)))
```

defaultdict(func, dict)
를 이용해서 새로운 element 가 insert 될 때,
log_missing() 함수를 call하는 코드.

새로 몇 개의 element 가 insert 되었는지는
log_missing() 함수 안에 nonlocal 변수로
체크할 수 있다. (책 참고)

23. 인터페이스가 간단하면 클래스 대신 함수를 받자.

```
# 별도의 작은 클래스를 만들어서 가독성을 향상시키는 방법
class CountMissing(object):
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0

counter = CountMissing()
result = None
result = defaultdict(counter.missing, current)

for key, amount in increments:
    result[key] += amount
print('added: {0}'.format(counter.added))
```

클래스 멤버 함수도 '함수' 이기 때문에
동일하게 적용이 가능하다.

클래스를 생성하는게 마음에 걸리지만..
(저 클래스는 뭘 의미하는거야? 라는 생각..)

23. 인터페이스가 간단하면 클래스 대신 함수를 받자.

```
class BetterCountMissing(object):
    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
counter()
assert callable(counter)

result = defaultdict(BetterCountMissing(), current)

print('Before: {0}'.format(dict(result)))
for key, amount in increments:
    result[key] += amount
print('added: {0}'.format(counter.added))
print('After: {0}'.format(dict(result)))
```

Callable class 를 만들어서 해결할 수 있다.

옆 코드에서 BetterCountMissing class는
함수 처럼 사용할 수 있다.

counter = BetterCountMissing()
counter() # 이게 가능
이걸 defaultdict 에 적용하면 왼쪽 모습이 된다.

23. 인터페이스가 간단하면 클래스 대신 함수를 받자.

책에 있는 정리

- 파이썬에서 컴포넌트 사이의 간단한 인터페이스용으로 클래스를 정의하고 인스턴스를 생성하는 대신에 함수만 써도 종종 충분하다.
- 파이썬에서 함수와 메서드에 대한 참조는 일급이다. 즉, 다른 타입처럼 표현식에서 사용할 수 있다.
- __call__ 이라는 특별한 메서드는 클래스의 인스턴스를 일반 파이썬 함수처럼 호출할 수 있게 해준다.
- 상태를 보존하는 함수가 필요할 때 상태 보존 클로저를 정의하는 대신 __call__ 메서드를 제공하는 클래스를 정의하는 방안을 고려하자.

23-1. Appendix. lambda

```
def make_incrementor(n):
    return lambda x: x - n

f = make_incrementor(2)
print(f(42))

print(make_incrementor(2)(42))  # 붙여서 사용하면 이런 모양


# 간단한 사용자 sort 예제
names = ['aaa', 'bbbb', 'cc', 'ddddddddd']
names.sort(key=lambda x: len(x))
print(names)

print(callable(f))
print(callable(make_incrementor))
print(callable(f(42)))
```

Q: print 될 결과를 예측해보세요.

A:
40
40
['cc', 'aaa', 'bbbb', 'ddddddddd']
True
True
False

Ref: http://www.secnetix.de/olli/Python/lambda_functions.hawk

23-2. Appendix. callable

callable(*object*)

Return `True` if the *object* argument appears callable, `False` if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

__call__() 을 가진 클래스나 일반 함수도 사용 가능

```
def test_function():
    return 1
print(callable(test_function))  # True
```

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

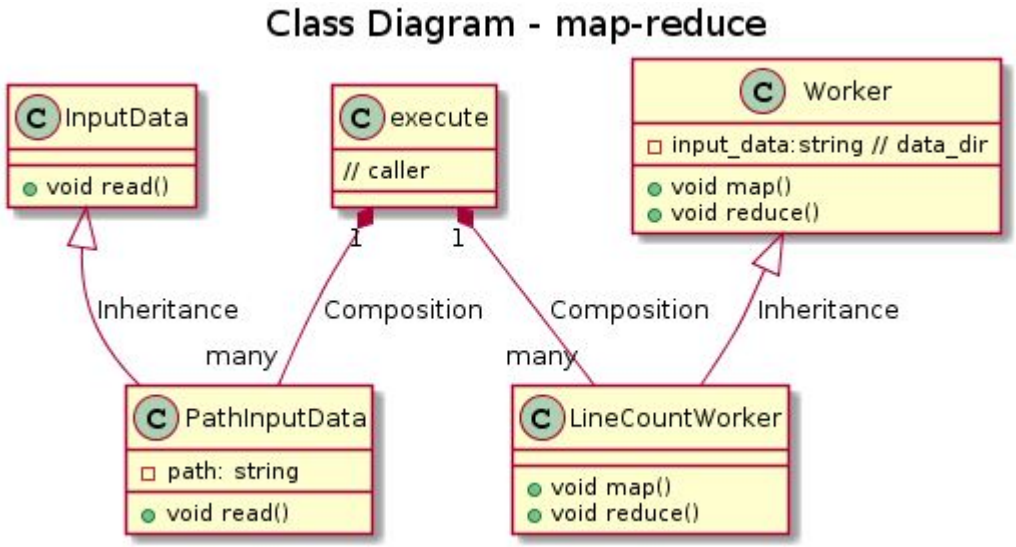
@classmethod?

Instance method	Class method
Instance가 있어야 호출 할 수 있다.	Class type 만으로 호출할 수 있다.
모든 멤버 함수는 self를 받아서 동작한다.	호출부의 타입이 무엇인가에 따라 원하는 타입의 함수를 찾아간다.

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

예제 코드가 깁니다..
uml 로 간단히..

코드는 pycharm을 봅시다.



24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

```python
# 위 클래스들을 생성하는 방법 예제
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))


def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

```python
def execute(workers):
    # MEMO: Thread header? from threading import Thread
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
    return first.result


# 순차적으로 호출
def map_reduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

호출부를 보면..

generate_inputs() 와 create_workers() 에서 추상화된 부모 클래스를 호출하지 못하고 최 하단 클래스를 호출하고 있음.

이렇게 되면? 클래스 타입이 바뀔때마다 수동으로 변경이 필요한 상황.
상속받은 것이 무색해짐.

이게 다..
다형성이 없어서 생기는 문제(python의 instance method 다형성은 생성자만 가능)

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

```
print('----------- about class method -----------')
class Parent(object):
    def __init__(self):
        print('parent.__init__')

    def work(self):
        print('parent.work')

    @classmethod
    def work_cls(cls):
        print('work_cls! {}'.format(cls))


class Child(Parent):
    def __init__(self):
        print('child.__init__')

    def work(self):
        print('child.work')

child_obj = Child()
child_obj.work()
child_obj.work_cls()

Parent.work_cls()  # work fine
# Parent.work()  # Error! not found instance
child_obj.work_cls()
```

성질 파악에는 역시 예제코드가 최고!

output:
child.__init__
child.work
work_cls! <class '__main__.Child'>
work_cls! <class '__main__.Parent'>
work_cls! <class '__main__.Child'>

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

```
with tempfile.TemporaryDirectory() as tmpdir:
    # write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = map_reduce(LineCountWorker, PathInputData, config)
```

호출부에서 타입 정보를 넘겨서
다른 클래스로 확장되어도
호출부만 수정하면 되도록 코드를 변경

(코드는 pycharm..)

24. 객체를 범용으로 생성하려면 @classmethod 다형성을 이용하자.

책에 있는 정리

- 파이썬에서는 클래스별로 생성자를 한 개(__init__ method)만 만들 수 있다.
- 클래스에 필요한 다른 생성자를 정의하려면 @classmethod를 사용하자.
- 구체 서브클래스들을 만들고 연결하는 범용적인 방법을 제공하려면 클래스 메서드 다형성을 이용하자.

25. super 로 부모 클래스를 초기화하자.

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value
        print("MyBaseClass.__init__() => {0}".format(self.value))


class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
        print("MyChildClass.__init__() => {0}".format(self.value))
```

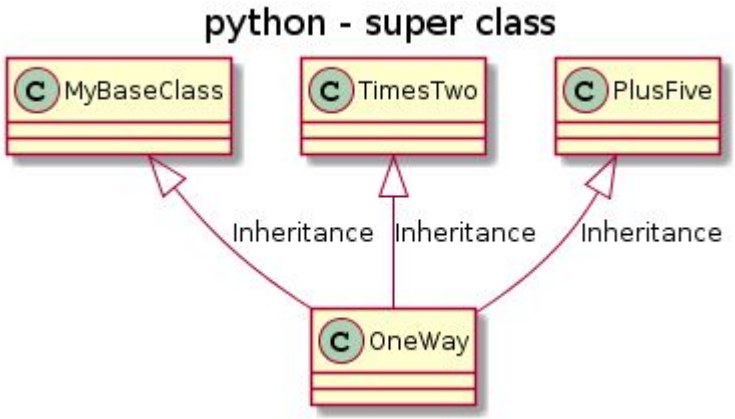
기본적으로 사용할 수 있는 호출

그러나 문제가 많다.

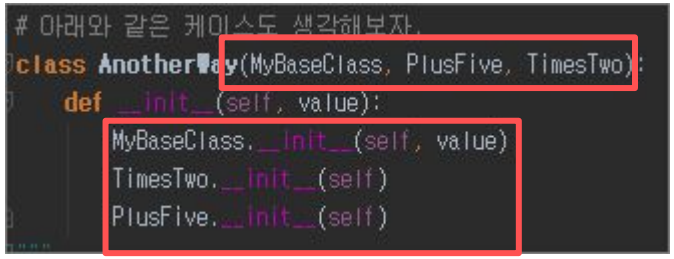
25. super 로 부모 클래스를 초기화하자.

```python
class TimesTwo(object):
    def __init__(self):
        self.value *= 2
        print("TimesTwo.__init__() => {0}".format(self.value))


class PlusFive(object):
    def __init__(self):
        self.value += 5
        print("PlusFive.__init__() => {0}".format(self.value))


# 위의 4가지 클래스를 이용하여 아래와 같은 클래스를 만들어보자.
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

1) 다중 상속 상황에서 발생하는 호출 순서 문제



25. super 로 부모 클래스를 초기화하자.



상속 받은 순서 vs 생성자를 호출한 순서
어떤 동작이 맞는 동작인가?

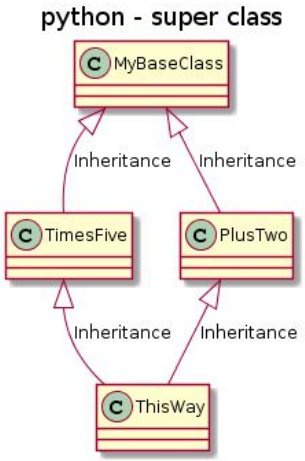
동작은? ⇒ **생성자를 호출한 순서!**

25. super 로 부모 클래스를 초기화하자.

2) 다이아몬드 상속(diamond inheritance)

: 호출 순서는 어떻게 될까?

```
ThisWay
        TimeFive
                MyBaseClass
                self.value = value
        value *= 5
        PlusTwo
                MyBaseClass #초기화
                self.value = value
        value += 2
result = ThisWay(5)
```




```python
class TimesFive(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 5
        print("TimesFive.__init__() => {0}".format(self.value))


class PlusTwo(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 2
        print("PlusTwo.__init__() => {0}".format(self.value))


class ThisWay(TimesFive, PlusTwo):
    def __init__(self, value):
        TimesFive.__init__(self, value)
        PlusTwo.__init__(self, value)

foo = ThisWay(5)
```

25. super 로 부모 클래스를 초기화하자.

이 문제를 해결하려고 ver 2.2 에서

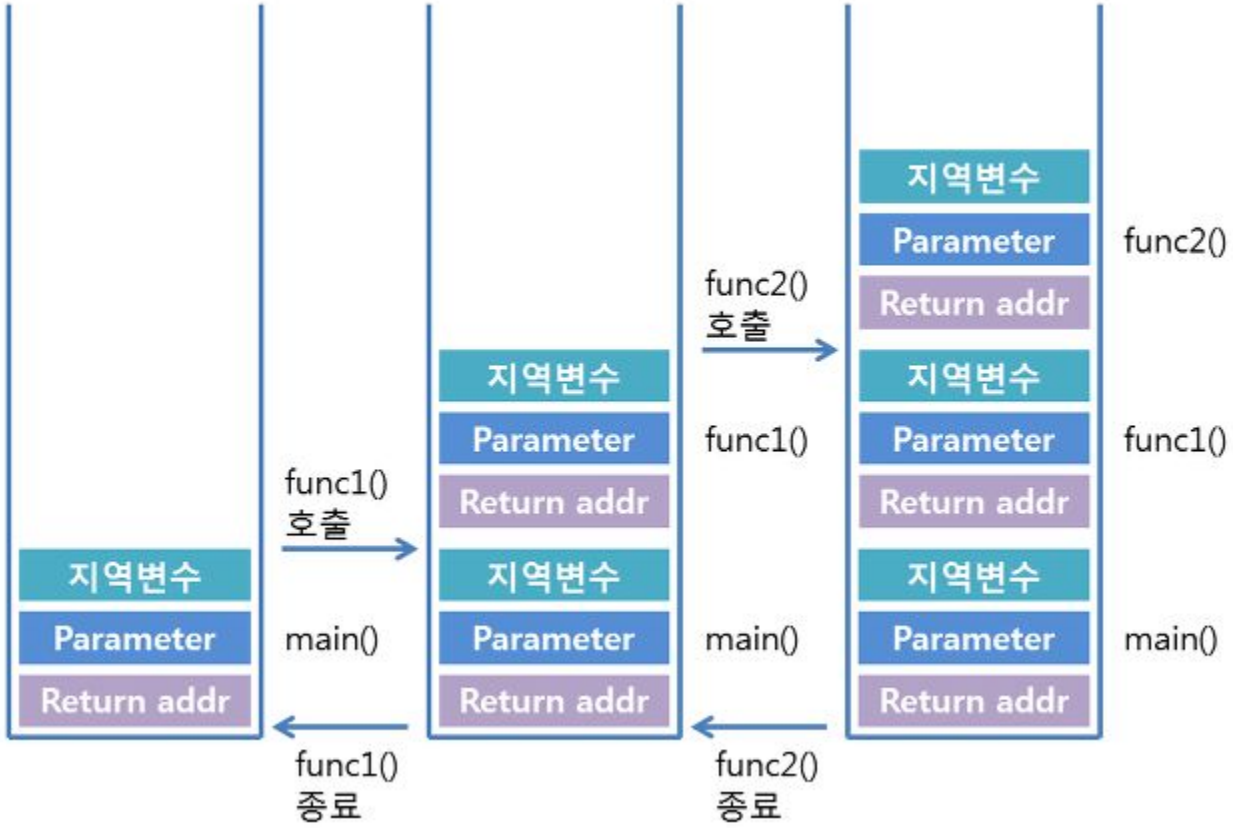
- super라는 내장 함수를 추가!
- MRO(Method Resolution Order, 메서드 해석 순서)를 정의!

Out:
MyBaseClass.__init__() => 5
Should be 5 * (5 + 2) = 35 and the result is 35
[<class '__main__.GoodWay'>,
 <class '__main__.TimesFiveCorrect'>,
 <class '__main__.PlusTwoCorrect'>,
 <class '__main__.MyBaseClass'>,
 <class 'object'>]
실제 연산은 함수를 빠져나오면서 한다.

```python
# 파이썬2
class TimesFiveCorrect(MyBaseClass):
    def __init__(self, value):
        super(TimesFiveCorrect, self).__init__(value)
        self.value *= 5


class PlusTwoCorrect(MyBaseClass):
    def __init__(self, value):
        super(PlusTwoCorrect, self).__init__(value)
        self.value += 2


class GoodWay(TimesFiveCorrect, PlusTwoCorrect):
    def __init__(self, value):
        super(GoodWay, self).__init__(value)


print('== python 2 sytle. ==')
foo = GoodWay(5)
print('Should be 5 * (5 + 2) = 35 and the result is {0}'.format(foo.value))
pprint(GoodWay.mro())
print(GoodWay.mro())
```

25. super 로 부모 클래스를 초기화하자.

이해돕기:



25. super 로 부모 클래스를 초기화하자.

파이썬 3.x 버전에서는
동일한 동작의 코드를 조금
축약해서 사용할 수 있다.

```
# 파이썬3
class Explicit(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value * 2)


class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)


assert Explicit(10).value == Implicit(10).value
```

25. super 로 부모 클래스를 초기화하자.

책에 있는 정리

- 파이썬의 표준 메서드 해석 순서(MRO)는 슈퍼클래스의 초기화 순서와 다이아몬드 상속 문제를 해결한다.
- 항상 내장 함수 super 로 부모 클래스를 초기화하자.

끗..