상 속

상속 형식

- 정의

```
class 기반 클래스:
    # 멤버 정의

class 파생클래스(기반 클래스)
    # 아무 멤버를 정의하지 않아도 기반 클래스의 모든 것을
      물려받아 갖게 됨.
    # 단, private 멤버(__로 시작되는 이름을 갖는 멤버)는 제외.
```

예)

```
class Base:
    def base_mehod(self):
        print("base_method")

class Derived(Base):
    pass
```

Derived 클래스는 상속을 통해 base_method() 메서드를 가짐.

Containment

- 형식

```
class A:
    def methodA(self):
        print("methodA() 호출")

class B:
    def __init__(self):
        self.instance_of_A = A()

    def call_methodA(self):
        self.instance_of_A.methodA()


if __name__ == "__main__":
    a = A()
    a.methodA()

    b = B()
    b.call_methodA()
```

다형성

- 형식

```
class ArmorSuite:
    def armor(self):
        print('armored')

class IronMan(ArmorSuite):
    pass

def get_armored(suite):
    suite.armor()

if __name__ == "__main__":
    suite = ArmorSuite()
    get_armored(suite)

    iron_man = IronMan()
    get_armored(iron_man)
```

상속 - 데이터 속성 주의

- 형식

```
class A:
    def __init__(self):
        print("A.__init__()")
        self.message = "Hello"

class B(A):
    def __init__(self):
        print("B.__init__()")

if __name__ == "__main__":
    obj = B()
    print(obj.message)              # error
```

상속 – super()

- 부모클래스의 객체 역할을 하는 프록시(Proxy)를 반환하는 내장 함수.

```
class A:
    def __init__(self):
        print(“A.__init__()")
        self.message = "Hello"

class B(A):
    def __init__(self):
        print("B.__init__()")

        super().__init__()
        print("self.message is "+self.message)

if __name__ == "__main__":
    b = B()
```

다중 상속

- 자식클래스 하나가 여러 부모 클래스로부터 상속받는 것.

```
class A:
    pass

class B:
    pass

class C:
    pass

class D(A, B, C):
    pass
```

다중 상속 - 주의

```
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")

class C(A):
    def method(self):
        print("C")

class D(B, C):
    pass

if __name__ == "__main__":
    obj = D()
    obj.method()              # 결과는 ?
```

오버라이딩(Overriding)

- 부모클래스로부터 상속받은 메서드를 다시 자식 클래스에 정의하는 것.

```
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")

class C(A):
    def method(self):
        print("C")

if __name == "__main__":
    A().method()        # A
    B().method()        # B
    C().method()        # C
```

데코레이터

- 함수를 꾸미는 객체.
- __call__() 메서드를 구현하는 클래스.
  - 객체를 함수 호출 방식으로 사용하게 만드는 마법 메서드.

```
class Callable:
    def __call__(self):
        print("I am called.")


if __name__ == "__main__":
    obj = Callable()
    obj()                   # 인스턴스 뒤에 괄호()를 붙여 호출하면,
                            # 내부적으로는 __call__() 메서드가 호출.
```

데코레이터-사용 방법1

- 생성자.

```
class MyDecorator:
    def __init__(self, f):                  # __init__() 메서드의 매개변수를 통해 함수를 받아들이고,
                                              데이터 속성에 저장해 둠.
        print("Initializing MyDecorator...")
        self.func = f                       # MyDecorator의 func 데이터 속성이 print_hello를 받아둠.

    def __call__(self):
        print("Begin : {0}".format(self.func.__name__))

        self.func()                         # __call()__ 메서드가 호출되면 생성자에서 저장해둔
                                              함수(데이터 속성)를 호출.
        print("End : {0}".format(self.func.__name__))


if __name__ == "__main__":
    def print_hello():
        print("Hello.")

    print_hello = MyDecorator(print_hello)
                                # MyDecorator의 인스턴스가 만들어지며 __init__() 메서드가 호출.
                                # print_hello 식별자는 앞에서 정의한 함수가 아닌 MyDecorator의 객체.

    print_hello()               # __call__() 메서드 덕에 MyDecorator 객체를 호출하듯 사용할 수 있음.
```

데코레이터-사용 방법2

- @ 기호.

```
class MyDecorator:
    def __init__(self, f):                    # __init__() 메서드의 매개변수를 통해 함수를 받아들이고,
                                                데이터 속성에 저장해 둠.
        print("Initializing MyDecorator...")
        self.func = f                         # MyDecorator의 func 데이터 속성이 print_hello를 받아둠.

    def __call__(self):
        print("Begin : {0}".format(self.func.__name__))

        self.func()                           # __call()__ 메서드가 호출되면 생성자에서 저장해둔
                                                함수(데이터 속성)를 호출.
        print("End : {0}".format(self.func.__name__))


if __name__ == "__main__":
    @MyDecorator
    def print_hello():
        print("Hello.")

    print_hello()
```

for문 순회 가능 객체 만들기

- 예) list

```
list = [1, 2, 3]
for e in list:
    print(e)
```

- range()

```
iterator = range(3).__iter__()
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())          # error
```

for문 순회 가능 객체 만들기

- 이터레이터(Iterator)

```
iterator = range(3).__iter__()
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())          # error
```

for문 순회 가능 객체 만들기

- 제너레이터(Generator)

```
def generator():
    yield 0
    yield 1
    yield 2
    yield 3


iterator = generator()
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
#print(iterator.__next__())    # error
```

상속의 조건 : 추상 기반 클래스

- 자식 클래스가 갖춰야 할 특징(메서드)을 강제하는 기능.
- 강제 조건 규약에 따르지 않으면 TypeError 예외 발생.
- metaclass=ABCMeta 클래스와 @abstractmethod 데코레이터를 이용.

```
from abc import ABCMeta
from abc import abstractmethod

class AbstractDuck(metaclass=ABCMeta):
    @abstractmethod
    def Quack(self):
        pass
```