

객체 지향 프로그래밍과 클래스

객체 지향 프로그래밍 - 객체와 클래스

❖ 객체(Object) = 속성(Attribute) + 기능(Method)

❖ 속성은 사물의 특징

- 예) 자동차의 속성 : 바디의 색, 바퀴의 크기, 엔진의 배기량

❖ 기능은 어떤 것의 특징적인 동작

- 예) 자동차의 기능 : 전진, 후진, 좌회전, 우회전

❖ 속성과 기능을 들어 자동차를 묘사하면?

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”

객체 지향 프로그래밍 - 객체와 클래스

❖ 다음과 같이 묘사한 자동차를 코드로 표현하면... (1)

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”

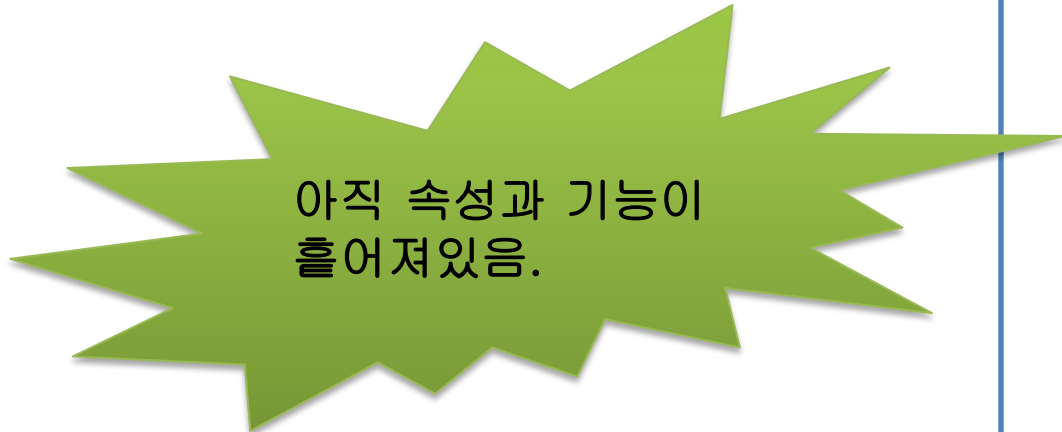
```
color = 0xFF0000    # 바디의 색
wheel_size = 16     # 바퀴의 크기
displacement = 2000 # 엔진 배기량
```

```
def forward(): # 전진
    pass
```

```
def backward(): # 후진
    pass
```

```
def turn_left(): # 좌회전
    pass
```

```
def turn_right(): # 우회전
    pass
```



아직 속성과 기능이
흘어져있음.

객체 지향 프로그래밍 - 객체와 클래스

❖ 다음과 같이 묘사한 자동차를 코드로 표현하면... (2)

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”

```
class Car:
```

Car 클래스의 정의 시작을 알립니다.

```
    def __init__(self):  
        self.color = 0xFF0000    # 바디의 색  
        self.wheel_size = 16     # 바퀴의 크기  
        self.displacement = 2000 # 엔진 배기량
```

Car 클래스 안에 차의 색, 바퀴 크기, 배기량을 나타내는 변수를 정의합니다.

```
    def forward(self): # 전진  
        pass
```

```
    def backward(self): # 후진  
        pass
```

```
    def turn_left(self): # 좌회전  
        pass
```

```
    def turn_right(self): # 우회전  
        pass
```

Car 클래스 안에 전진, 후진, 좌회전, 우회전 함수를 정의합니다.

객체 지향 프로그래밍 - 객체와 클래스

❖ 앞에서 만든 Car 클래스는 자료형

- Car 클래스의 객체는 다음과 같이 정의함

```
num = 123      # 자료형:int, 변수: num  
my_car = Car() # 자료형: Car 클래스, 객체: my_car
```

❖ 객체 대신 인스턴스(Instance)라는 용어를 사용하기도 함.

- 클래스가 설계도, 객체는 그 설계를 바탕으로 실체화한 것이라는 뜻에서 유래한 용어
- 객체뿐 아니라 변수도 인스턴스라고 부름. 자료형을 메모리에 실체화한 것이 변수이기 때문임.

객체 지향 프로그래밍 - 파이썬에서 코딩하며 객체를 지향한다.

❖ 컴퓨터의 업그레이드가 용이한 이유?

- 컴퓨터 부품간의 결합도(coupling)가 낮기 때문.

❖ 이와 비해 태블릿과 스마트폰의 업그레이드는 거의 불가능.

- 부품간의 결합도가 매우 높기 때문.

❖ 결합도는 한 시스템 내의 구성 요소간의 의존성을 나타내는 용어.

- 소프트웨어에서도 결합도가 존재함.
- 예) A() 함수를 수정했을 때 B() 함수의 동작에 부작용이 생긴다면 이 두 함수는 강한 결합도를 보인다고 할 수 있음.
- 예) A() 함수를 수정했는데도 B() 함수가 어떤 영향도 받지 않는다면 이 두 함수는 약한 결합으로 이루어져 있다고 할 수 있음.

❖ 클래스 안에 같은 목적과 기능을 위해 묶인 코드 요소(변수, 함수)는 객체 내부에서만 강한 응집력을 발휘하고 객체 외부에 주는 영향은 줄이게 됨.

클래스의 정의

❖ 클래스는 다음과 같이 **class** 키워드를 이용하여 정의.

```
class 클래스이름:  
    코드블록
```

❖ 클래스의 코드블록은 변수와 메소드(Method)로 이루어짐.

- 기능(Method) : 객체 지향 프로그래밍에서 사물의 동작을 나타냄.
- 메소드(Method) : 객체 지향 프로그래밍의 기능에 대응하는 파이썬 용어. 함수와 거의 동일한 의미이지만 메소드는 클래스의 멤버라는 점이 다름.
- 함수(Function) : 일련의 코드를 하나의 이름 아래 묶은 코드 요소.

❖ 객체의 멤버(메소드와 데이터 속성에 접근하기)

- 객체의 멤버에 접근할 때는 점(.)을 이용.

```
my_car = Car()  
print( my_car.color )
```

my_car의 멤버에 접근하게 해줍니다.

클래스의 정의

❖ 예제 : 09/Car.py

```
class Car:
    def __init__(self):
        self.color = 0xFF0000    # 바디의 색
        self.wheel_size = 16     # 바퀴의 크기
        self.displacement = 2000 # 엔진 배기량

    def forward(self): # 전진
        pass

    def backward(self): # 후진
        pass

    def turn_left(self): # 좌회전
        pass

    def turn_right(self): # 우회전
        pass
```


클래스의 정의

❖ 예제 : 09/Car.py

#Car 클래스 정의 종료. 아래는 Car 클래스의 인스턴스를 정의하고 사용하는 코드

```
if __name__ == '__main__':  
    my_car = Car()  
  
    print('0x{:02X}'.format(my_car.color))  
    print(my_car.wheel_size)  
    print(my_car.displacement)  
  
    my_car.forward()  
    my_car.backward()  
    my_car.turn_left()  
    my_car.turn_right()
```

○ 실행결과

```
>Car.py  
0xFF0000  
16  
2000
```

클래스의 정의 - __init__() 메소드를 이용한 초기화

❖ __init__()

- 객체가 생성된 후 가장먼저 호출되는 메소드
- "초기화하다"는 뜻의 initialize를 줄여서 붙여진 이름.

❖ 예제 : 09/InstanceVar.py

```
class InstanceVar:
    def __init__(self):
        self.text_list = []

    def add(self, text):
        self.text_list.append(text)

    def print_list(self):
        print(self.text_list)

if __name__ == '__main__':
    a = InstanceVar()
    a.add('a')
    a.print_list() # ['a'] 출력을 기대

    b = InstanceVar()
    b.add('b')
    b.print_list() # ['b'] 출력을 기대
```


• 실행 결과

```
>InstanceVar.py
['a']
['b']
```

클래스의 정의 - __init__() 메소드를 이용한 초기화

❖ 매개변수를 입력받는 __init__() 메소드의 예

```
class ContactInfo:  
    def __init__(self, name, email):  
        self.name = name  
        self.email = email  
  
sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')
```



The diagram illustrates the parameter passing in the `__init__` method. Two orange boxes labeled 'name' and 'email' are positioned to the right of the `self.name = name` and `self.email = email` lines, respectively. Arrows point from these boxes to the `name` and `email` parameters in the `__init__` method signature.

클래스의 정의 - __init__() 메소드를 이용한 초기화

❖ 예제 : 09/ContactInfo.py

```
class ContactInfo:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def print_info(self):
        print('{0} : {1}'.format(self.name, self.email))

if __name__ == '__main__':
    sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')
    hanbit = ContactInfo('hanbit', 'noreply@hanb.co.kr')

    sanghyun.print_info()
    hanbit.print_info()
```

```
>ContactInfo.py
박상현 : seanlab@gmail.com
hanbit : noreply@hanb.co.kr
```

클래스의 정의 - self에 대하여

- ❖ 파이썬의 메소드에 사용되는 self가 가리키는 “자신”은 바로 메소드가 소속되어 있는 객체

```
class ContactInfo:
    def __init__(self, name, email):
        self.name = name
        self.email = email

sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')
```

- ❖ ContactInfo 외부에서는 sanghyun이라는 이름으로 객체를 다룰 수 있음.
- ❖ 내부에서는 sanghyun처럼 객체를 지칭할 수 있는 이름이 없기 때문에 self가 도입되었음.

클래스의 정의 - 정적 메소드와 클래스 메소드

❖ 인스턴스 메소드 - 인스턴스(객체)에 속한 메소드

- 인스턴스 메소드가 "인스턴스에 속한다"라는 표현은 "인스턴스를 통해 호출 가능하다."라는 뜻

❖ 정적 메소드와 클래스 메소드는 클래스에 귀속

❖ 정적 메소드

- @staticmethod 데코레이터로 수식
- self 키워드 없이 정의

```
class 클래스이름:
```

```
    @staticmethod
```

```
    def 메소드이름( 매개변수 ):
```

```
        pass
```

@staticmethod 데코레이터로 수식합니다.

self 매개변수는 사용하지 않습니다.

클래스의 정의 - 정적 메소드와 클래스 메소드

❖ 예제 : 09/Calculator.py(정적 메소드)

```
class Calculator:
    @staticmethod
    def plus(a, b):
        return a+b

    @staticmethod
    def minus(a, b):
        return a-b

    @staticmethod
    def multiply(a, b):
        return a*b

    @staticmethod
    def divide(a, b):
        return a/b

if __name__ == '__main__':
    print("{0} + {1} = {2}".format(7, 4, Calculator.plus(7, 4)))
    print("{0} - {1} = {2}".format(7, 4, Calculator.minus(7, 4)))
    print("{0} * {1} = {2}".format(7, 4, Calculator.multiply(7, 4)))
    print("{0} / {1} = {2}".format(7, 4, Calculator.divide(7, 4)))
```

클래스의 정의 - 정적 메소드와 클래스 메소드

- 실행결과

```
>Calculator.py
```

```
7 + 4 = 11
```

```
7 + 4 = 3
```

```
7 + 4 = 28
```

```
7 + 4 = 1.75
```


클래스의 정의 - 정적 메소드와 클래스 메소드

❖ 클래스 메소드

- @classmethod 데코레이터로 수식
- cls 매개변수 사용

```
class 클래스이름:  
    # ...
```

```
    @classmethod  
    def 메소드이름(cls):  
        pass
```

클래스 메소드를 정의하기 위해서는...
1. @classmethod 데코레이터를 앞에 붙여줍니다.

2. 메소드의 매개변수를 하나 이상 정의합니다.

클래스의 정의 - 정적 메소드와 클래스 메소드

❖ 실습 1

```
>>> class TestClass:
    @classmethod
    def print_TestClass(cls):
        print(cls)
```

```
>>> TestClass.print_TestClass()
```

클래스를 통한 클래스 메소드 호출

```
<class '__main__.TestClass'>
```

```
>>> obj = TestClass()
```

```
>>> obj.print_TestClass()
```

인스턴스를 통한 클래스 메소드 호출

```
<class '__main__.TestClass'>
```

클래스의 정의 - 정적 메소드와 클래스 메소드

❖ 예제 : 09/InstanceCounter.py(클래스 메소드_

```
class InstanceCounter:
    count = 0
    def __init__(self):
        InstanceCounter.count += 1
```

```
    @classmethod
    def print_instance_count(cls):
        print(cls.count)
```

print_instance_count() 메소드는 InstanceCounter의 클래스 변수인 count를 출력합니다.

```
if __name__ == '__main__':
    a = InstanceCounter()
    InstanceCounter.print_instance_count()

    b = InstanceCounter()
    InstanceCounter.print_instance_count()

    c = InstanceCounter()
    c.print_instance_count()
```

○ 실행 결과

```
>InstanceCounter.py
1
2
3
```

클래스의 정의 - 클래스 내부에게만 열려있는 프라이빗 멤버

- ❖ 클래스도 코드 블록을 가지므로 “안”과 “밖” 개념이 존재함
- ❖ 다음 코드에서 MyClass 클래스의 “안”은 상자로 표시한 부분이고 “밖”은 상자로 표시된 부분을 제외한 나머지 코드를 말함.

```
class YourClass:
    pass

class MyClass:
    def __init__(self):
        self.message = "Hello"

    def some_method(self):
        print(self.message)

obj = MyClass()
obj.some_method()
```

- 프라이빗(Private) 멤버 : 클래스 내부에서만 접근이 가능한 멤버
- 퍼블리(Public) 멤버 : 안과 밖 모두에서 접근이 가능한 멤버

클래스의 정의 - 클래스 내부에게만 열려있는 프라이빗 멤버

❖ 프라이빗 멤버 명명 규칙

- 두 개의 밑줄 `__` 이 접두사여야 한다. 예) `__number`
- 접미사는 밑줄이 한 개까지만 허용된다. 예) `__number_`
- 접미사의 밑줄이 두 개 이상이면 퍼블릭 멤버로 간주합니다. 예) `__number__`

클래스의 정의 - 클래스 내부에게만 열려있는 프라이빗 멤버

❖ 프라이빗 멤버 명명 규칙

```
>>> class HasPrivate:
    def __init__(self):
        self.public = "Public."
        self.__private = "Private."

    def print_from_internal(self):
        print(self.public)
        print(self.__private)
```

```
>>> obj = HasPrivate()
>>> obj.print_from_internal()
Public.
Private.
```

```
>>> print(obj.public)
Public.
```

```
>>> print(obj.__private)
Traceback (most recent call last):
```

```
File "<pyshell#46>", line 1, in <module>
```

```
    print(obj.__private)
```

```
AttributeError: 'HasPrivate' object has no attribute '__private'
```

HasPrivate의 print_from_internal() 함수는 public, __private 두 데이터 속성에 자유롭게 접근할 수 있습니다. 같은 클래스의 멤버끼리니까요.

HasPrivate 객체 외부에서는 __private 데이터 속성에 접근할 수 없습니다. __private 데이터 속성이 아예 존재하지 않는 것처럼 보이기 때문입니다.

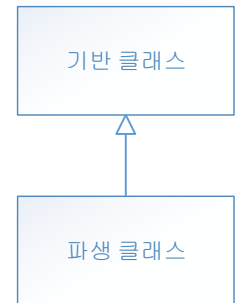
상속

❖ “상속(Inheritance)”

❖ 한 클래스가 다른 클래스로부터 데이터 속성과 메소드를 물려받는 것.

```
class 기반 클래스:  
    # 멤버 정의
```

```
class 파생 클래스(기반 클래스)  
# 아무 멤버를 정의하지 않아도 기반 클래스의 모든 것을 물려받아 갖게 됩니다.  
# 단, 프라이빗 멤버(__로 시작되는 이름을 갖는 멤버)는 제외입니다.
```



상속

❖ 실습

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

>>> class Derived(Base):
    pass

>>> base = Base()
>>> base.base_method()
base_method
>>> derived = Derived()
>>> derived.base_method()
base_method
```


상속 – super()

- ❖ **super()는 부모 클래스의 객체 역할을 하는 프록시(Proxy)를 반환하는 내장함수**
 - super() 함수의 반환 값을 상위클래스의 객체로 간주하고 코딩.
 - 객체 내의 어떤 메소드에서든 부모 클래스에 정의되어 있는 버전의 메소드를 호출하고 싶으면 super()를 이용.
- ❖ **예제 : 09/super.py**

```
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()
```

- 실행 결과

```
>super.py
B.__init__()
A.__init__()
self.message is Hello
```

상속 - 다중 상속

❖ 다중상속은 자식 하나가 여러 부모(?!)로부터 상속을 받는 것

- 파생 클래스의 정의에 기반 클래스의 이름을 콤마(,)로 구분해서 꼭 적어 주면 다중상속이 이루어짐.

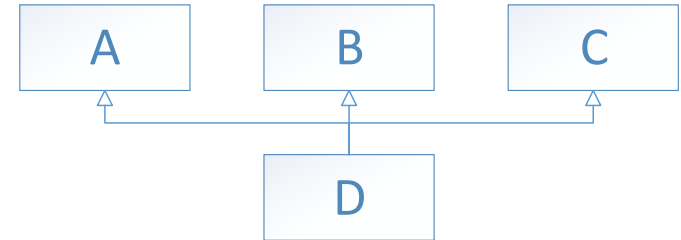
```
class A:  
    pass
```

```
class B:  
    pass
```

```
class C:  
    pass
```

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

클래스 D는 클래스 A, B, C를 부모로
부터 상속받습니다.



상속 - 다중 상속

❖ 다이아몬드 상속 : 다중 상속이 만들어 내는 곤란한 상황

- D는 B와 C 중 누구의 method()를 물려받게 되는 걸까?

```
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
```

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

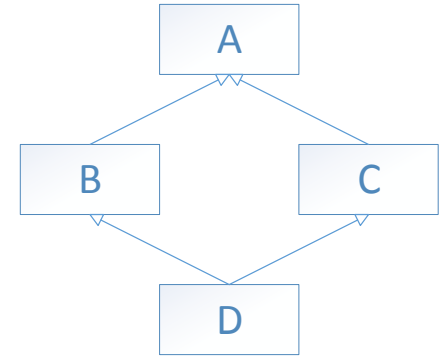
>>> class B(A):
    def method(self):
        print("B")

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

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

```
>>> obj = D
>>> obj.method()
B
```

D는 B의 method()를 물려받았습니다.



상속 - 오버라이딩

❖ OOP에서 오버라이딩의 뜻은 “기반(부모) 클래스로부터 상속받은 메소드를 다시 정의하다.”

❖ 실습 1

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

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

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

```
>>> A().method()
```

```
A
```

```
>>> B().method()
```

```
B
```

```
>>> C().method()
```

```
C
```

B는 A를 상속하지만, A의 method()를 물려받는 대신 자신만의 버전을 재정의(오버라이딩) 함.

C도 A를 상속하지만, A의 method()를 물려받는 대신 자신만의 버전을 재정의(오버라이딩) 함.

데코레이터 : 함수를 꾸미는 객체

❖ 데코레이터는 `__call__()` 메소드를 구현하는 클래스

- `__call__()` 메소드는 객체를 함수 호출 방식으로 사용하게 만드는 마법 메소드

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

```
>>> obj = Callable()  
>>> obj()  
I am called.
```

인스턴스 뒤에 괄호 (와)를 붙여 "호출" 하면, 내부적으로는 `__call__` 메소드가 호출됩니다.

데코레이터 : 함수를 꾸미는 객체

❖ 예제 : 09/decorator1.py(데코레이터 선언과 사용 1)

```
class MyDecorator:
    def __init__(self, f):
        print("Initializing MyDecorator...")
        self.func = f

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

def print_hello():
    print("Hello.")
```

MyDecorator의 func 데이터 속성이 print_hello를 받아둡니다.

```
print_hello = MyDecorator(print_hello)
```

MyDecorator의 인스턴스를 만들어지며 __init__() 메소드가 호출됩니다. print_hello 식별자는 앞에서 정의한 함수가 아닌 MyDecorator의 객체입니다.

```
print_hello()
```

__call__() 메소드 덕에 MyDecorator 객체를 호출하듯 사용할 수 있습니다.

○ 실행 결과:

```
>decorator1.py
Initializing MyDecorator...
Begin :print_hello
Hello.
End :print_hello
```

데코레이터 : 함수를 꾸미는 객체

❖ 예제 : 09/decorator2.py(데코레이터 선언과 사용 2)

```
class MyDecorator:
    def __init__(self, f):
        print("Initializing MyDecorator...")
        self.func = f

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

@MyDecorator
def print_hello():
    print("Hello.")

print_hello()
```

○ 실행 결과:

```
>decorator2.py
Initializing MyDecorator...
Begin :print_hello
Hello.
End :print hello
```

for 문으로 순회를 할 수 있는 객체 만들기

- 이터레이터와 순회 가능한 객체

❖ 파이썬에서 for문을 실행할 때 가장먼저 하는 일은 순회하려는 객체의 `__iter__()` 메소드를 호출하는 것.

- `__iter__()` 메소드는 이터레이터(Iterator)라고 하는 특별한 객체를 for 문에게 반환(이터레이터는 `__next__()` 메소드를 구현하는 객체)
- for문은 매 반복을 수행할 때마다 바로 이 `__next__()` 메소드를 호출하여 다음 요소를 얻어냄.

❖ `range()` 함수가 반환하는 객체도 순회가능한 객체.

❖ 실습 1

```
>>> iterator = range(3).__iter__()
>>> iterator.__next__()
0
>>> iterator.__next__()
1
>>> iterator.__next__()
2
>>> iterator.__next__()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    iterator.__next__()
StopIteration
```


for 문으로 순회를 할 수 있는 객체 만들기

- 이터레이터와 순회 가능한 객체

❖ 예제 : 09/iterator.py(직접 구현한 range() 함수)

```
class MyRange:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.end:
            current = self.current
            self.current += 1
            return current
        else:
            raise StopIteration()

for i in MyRange(0, 5):
    print(i)
```

■ 실행 결과

0
1
2
3
4

for 문으로 순회를 할 수 있는 객체 만들기 - 제네레이터

- ❖ 제네레이터(Generator)는 yield문을 이용하여 이터레이터보다 더 간단한 방법으로 순회가능한 객체를 만들게 해줌.
 - yield문은 return문처럼 함수를 실행하다가 값을 반환하지만, return문과는 달리 함수를 종료시키지는 않고 중단시켜놓기만함.
- ❖ 예제 : 09/generator.py

```
def YourRange(start, end):  
    current = start  
    while current < end:  
        yield current  
        current += 1  
    return  
  
for i in YourRange(0, 5):  
    print(i)
```

- 실행 결과

```
0  
1  
2  
3  
4
```

상속의 조건 : 추상 기반 클래스

- ❖ 추상 기반 클래스(Abstract Base Class)는 자식 클래스가 갖춰야 할 특징 (메소드)을 강제
 - 추상 기반 클래스를 정의할 때는 다음과 같이 abc 모듈의 ABCMeta 클래스와 @abstractmethod 데코레이터를 이용

상속의 조건 : 추상 기반 클래스

❖ 실습 1 (ABC의 규칙을 위반했을 때)

```
>>> from abc import ABCMeta
>>> from abc import abstractmethod
>>> class
AbstractDuck(metaclass=ABCMeta):
    @abstractmethod
    def Quack(self):
        pass

>>> class Duck(AbstractDuck):
    pass

>>> duck = Duck()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in
<module>
    duck = Duck()
TypeError: Can't instantiate abstract
class Duck with abstract methods Quack
```

❖ 실습 2 (ABC의 규칙을 준수)

```
>>> class Duck(AbstractDuck):
    def Quack(self):
        print("[Duck] Quack")

>>> duck = Duck()
>>> duck.Quack()
[Duck] Quack
```

상속의 조건 : 추상 기반 클래스

- ❖ 추상 기반 클래스(Abstract Base Class)는 자식 클래스가 갖춰야 할 특징(메소드)을 강제
 - 추상 기반 클래스를 정의할 때는 다음과 같이 abc 모듈의 ABCMeta 클래스와 @abstractmethod 데코레이터를 이용
- ❖ 실습 1 (ABC의 규칙을 위반했을 때)