클래스(Class)

안화수

1. 클래스

- class: 관련있는 저장공간과 기능을 하나로 묶은 것 - Encapsulation
- class Object: 클래스와 동일한 의미로 사용하는데 어떤 클래스를 구체적으로 지정하기 위해 사용
- class instance: 클래스를 호출하여 생성된 객체
- method: 클래스 안에 정의된 함수
- member variable: 클래스 안에 정의된 변수
- attribute: 클래스 안에 있는 모든 것
- Inheritance: 하위 클래스가 상위 클래스의 모든 속성을 물려받는 것
- Super Class: Base Class라고도 하는데 다른 클래스의 상위 클래스
- Sub Class: Derived Class라고도 하는데 다른 클래스로부터 속성을 물려받는 클래스
- Multiple Inheritance: 2개 이상의 클래스로부터 상속받는 것
- Polymorphism: 동일한 코드가 상황에 따라 다르게 반응하는 것

1. 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."

1. 클래스

- 다음과 같이 묘사한 자동차를 코드로 표현
  - "18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다."

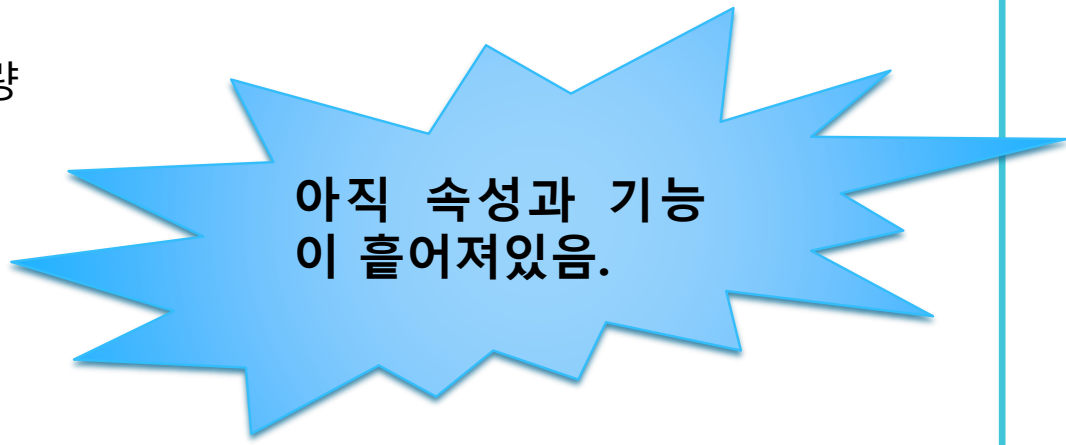
```
color = 0xFF0000     # 바디의 색
wheel_size = 16      # 바퀴의 크기
displacement = 2000 # 엔진 배기량

def forward(): # 전진
    pass

def backward(): # 후진
    pass

def turn_left(): # 좌회전
    pass

def turn_right(): # 우회전
    pass
```



1. 클래스

- 다음과 같이 묘사한 자동차를 코드로 표현하면... (2)
  - "18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다."

```
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
```

Car 클래스의 정의 시작을 알립니다.

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

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

1. 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.

1. 클래스

- 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 작성해야 합니다.
- 클래스를 잘 설계하면 유지보수가 편리해지고 재사용성이 높아집니다.

1. 클래스

- 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.

2. 메소드

- 메소드 생성
  - 메소드를 클래스 내부에 선언할 때는 첫번째 매개변수는 무조건 현재 클래스의 객체가 되어야 합니다.
  - 관습적으로 self 라는 단어를 이용합니다.
  - 두번째 매개변수 부터는 사용자가 정의할 수 있습니다.
  - 메소드를 호출하는 방법
    - 클래스 이름을 이용한 호출(언바운드 호출)

      클래스이름.메소드이름(인스턴스이름, 매개변수)
    - 인스턴스 이름을 이용한 호출(바운드 호출)

      인스턴스이름.메소드이름(매개변수)
    - 클래스 내부에서 자신의 클래스에 속한 메소드를 호출

      self.메소드이름(매개변수)

3. 변수

❖ Student.py 파일을 생성하고 작성

```
class student:
    def start(self):
        print("안녕하세요")
    def printName(self, name):
        print("이름은 {0}".format(name))
```

3. 변수

❖ __main__.py 파일을 생성하고 작성

```
from Student import student
stu = student()
student.start(stu)
stu.printName("중앙")
```

3. 변수

- 클래스 내의 변수 생성
  - 메소드 외부에서 선언하면 클래스 변수: 클래스 내부에 1개만 만들어지며 클래스 이름으로 접근할 수 있고 객체 이름으로도 접근해서 사용할 수 있게 됩니다.
  - 메소드 내부에서 self와 함께 선언되면 인스턴스 변수: 각 객체 내부에 각각 만들어지면 객체 이름으로만 접근해서 사용할 수 있습니다.
  - 메소드 내부에서 self. 과 함께 변수를 선언하지 않으면 그 변수는 메소드의 지역변수가 됩니다.

3. 변수

❖ Student.py 파일을 수정 – 클래스 변수 생성

```
class student:
    schoolName = "Korea"
```

3. 변수

❖ __main__.py 파일을 수정

```
from Student import student
stu1 = student()
stu2 = student()
print("stu1의 주소:{0}".format(id(stu1)))
print("stu2의 주소:{0}".format(id(stu2)))
#위 2개의 객체는 각각 생성자를 호출해서 만들어 진 것이므로 서로 다른 영역을 차지하고 만들어집니다.
student.schoolName="Seoul"
'''
schoolName은 메소드 외부에 만들어진 것이므로 class 변수가 되고
클래스 변수는 1개만 만들어서
클래스와 클래스로부터 만들어진 객체 모두가 공유해서 사용합니다.
따라서 아래 3개의 출력문은 모두 동일한 값을 출력합니다.
'''
print("Student의 학교:{0}".format(student.schoolName))
print("stu1의 학교:{0}".format(stu1.schoolName))
print("stu2의 학교:{0}".format(stu2.schoolName))
```

3. 변수

❖ Student.py 파일을 수정 – 인스턴스 변수 생성

```
class student:
    schoolName = "Korea"
    def setName(self, name):
        self.name = name
    def getName(self):
        return self.name
```

3. 변수

❖ __main__.py 파일을 수정

```
from Student import student
stu1 = student()
stu2 = student()
stu1.setName("Steve Jobs")
stu2.setName("Steve wozniak")
print("stu1의 이름:{0}".format(stu1.getName()))
print("stu2의 이름:{0}".format(stu2.getName()))
```

4. 생성자와 소멸자

- 생성자: __init__()
  - 객체가 생성된 후 가장 먼저 호출되는 메소드
  - "초기화하다"는 뜻의 initialize를 줄여서 붙여진 이름
  - 첫번째 매개변수는 self 이며 이후에 매개변수 추가 가능
  - self 이외의 매개변수가 있는 생성자를 만들면 인스턴스를 생성할 때 매개변수를 넘겨주어야 합니다.
  - 멤버 변수의 초기화 코드를 주로 작성합니다.

- 소멸자:__del__()
  - 객체가 소멸될 때 호출되는 메소드
  - 외부 자원을 사용하는 경우 해제하는 코드를 주로 작성
  - self 이외의 매개변수를 받지 않습니다.

4. 생성자와 소멸자

❖ __main__.py 파일을 수정

```
from Student import student
stu1 = student()
stu2 = student()
print("stu1의 이름:{0}".format(stu1.getName()))
print("stu2의 이름:{0}".format(stu2.getName()))
```

위처럼 프로그램을 실행시키면 에러
setName을 호출하기 전에 getName을 호출해서 아직 name이 만들어지기 전 상태

```
Traceback (most recent call last):
  File "C:□Users□Administrator□python□test□__main__.py", line 4, in <module>
    print("stu1의 이름:{0}".format(stu1.getName()))
  File "C:□Users□Administrator□python□test□Student.py", line 6, in getName
    return self.name
AttributeError: 'student' object has no attribute 'name'
```

4. 생성자와 소멸자

❖ Student.py 파일을 수정

```
from time import ctime
class student:
    #생성자
    def __init__(self, name="noname"):
        print("{0}에 객체가 생성되었습니다.".format(ctime()))
        self.name=name

    schoolName = "Korea"
    def setName(self, name):
        self.name = name
    def getName(self):
        return self.name

    #소멸자
    def __del__(self):
        print("{0}에 객체가 소멸되었습니다.".format(ctime()))
```

4. 생성자와 소멸자

❖ __main__.py 파일을 수정

```
from Student import student
stu1 = student("중앙")
stu2 = student()
print("stu1의 이름:{0}".format(stu1.getName()))
print("stu2의 이름:{0}".format(stu2.getName()))
del stu1
del stu2
```

5.정적 메소드&클래스 메소드

- 정적 메소드
  - 인스턴스를 생성하지 않고 클래스를 이용해서 직접 호출할 수 있는 메소드
  - 메소드 내에서 멤버 변수를 호출할 수 없습니다.
  - 클래스 변수는 호출할 수 있습니다.
  - @staticmethod 데코레이터로 수식
  - self 키워드 없이 정의

```
class 클래스이름:
    @staticmethod
    def 메소드이름( 매개변수 ):
        pass
```

@staticmethod데코레이터로 수식합니다.

self 매개변수는 사용하지 않습니다.

5.정적 메소드&클래스 메소드

- 클래스 메소드
  - @classmethod 데코레이터로 수식
  - 정적 메소드와 유사하지만 첫번째 매개변수로 클래스 객체가 전달되는 것이 다릅니다.
  - 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.

5.정적 메소드&클래스 메소드

❖ Student.py 파일을 수정

```
class student:
    @classmethod
    def cmethod(cls):
        print("클래스 메소드")
        print(cls)

    @staticmethod
    def smethod():
        print("정적 메소드")
```

5.정적 메소드&클래스 메소드

❖ __main__.py 파일을 수정

```
from Student import student
student.cmethod()
student.smethod()
```

6.Property

- private 멤버 명명 규칙
  - private 멤버는 클래스 내부에서는 접근이 가능하지만 클래스 외부(인스턴스)에서 접근이 되지 않는 멤버
  - 이와 반대되는 개념으로 public이 있는데 public 멤버는 클래스 외부(인스턴스)에서 접근이 가능한 멤버입니다.
  - python의 모든 멤버는 기본적으로 public
  - 문법적으로 지원하지는 않고 이름을 이용해서 private 이라는 의미만 전달
  - 두 개의 밑줄 __ 이 접두사여야 합니다.
  - 접미사는 밑줄이 한 개까지만 허용됩니다.
  - 접미사의 밑줄이 두 개 이상이면 public으로 간주합니다.
- Property
  - 변수를 호출하는 것 처럼 사용하지만 실제로는 getter 와 setter 메소드를 호출하는 것으로 처리되는 속성
  - 변수명 = property(fget=None, fset=None, fdel=None, doc=None)
  - fget은 getter, fset은 setter, fdel은 삭제할 때 사용할 메소드

5.정적 메소드&클래스 메소드

❖ Student.py 파일을 수정

```
class student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        print("setter 호출")
        self.__name = name

    def getName(self):
        print("getter 호출")
        return self.__name
```

5.정적 메소드&클래스 메소드

❖ __main__.py 파일을 수정

```
from Student import student
stu = student()
#아래 문장은 변수를 호출한 것으로 메소드 호출문은 아닙니다.
stu.name = "kim"
print(stu.name)
```

5.정적 메소드&클래스 메소드

- Student.py 파일을 수정: 아래처럼 수정하면 변수를 호출하는 코드를 작성해도 getter와 setter 메소드를 호출합니다.

```
class student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        print("setter 호출")
        self.__name = name

    def getName(self):
        print("getter 호출")
        return self.__name
    name = property(getName, setName)
```

상속(Inheritance)

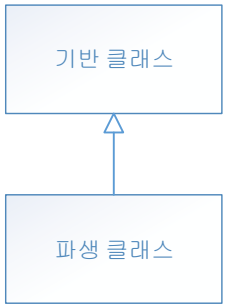
안 화 수

상속

- 상속(Inheritance)
  - 한 클래스가 다른 클래스로부터 데이터 속성과 메소드를 물려받는 것
  - 상속하는 클래스를 기반(Base)클래스 또는 상위(Super)클래스라고 하고 상속을 받는 클래스를 파생(Derived)클래스, 하위(Sub)클래스라고 합니다.

```
class 기반 클래스:
    # 멤버 정의


class 파생 클래스(기반 클래스)
# 아무 멤버를 정의하지 않아도 기반 클래스의 모든 것을 물려받아 갖게 됩니다.
# private 멤버(__로 시작되는 이름을 갖는 멤버)는 상속은 되지만 접근할 수
없습니다..
```



상속

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


class Derived(Base):
    pass

base = Base()
base.base_method()
derived = Derived()
derived.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 메소드를 호출하고 싶으면 super()를 이용.
- 예제

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

class derived(base):
    def __init__(self):
        print("B.__init__()")

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

ob = derived()
```

- 실행 결과

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

상속 – Overriding(재정의)

- Overriding은 부모 클래스에 있는 메소드를 하위 클래스에서 다시 정의하는 것
- 상위 클래스의 기능이 부족해서 기능을 추가하기 위한 목적과 다형성 구현을 위해서 합니다.
- 자신이 만든 클래스가 아닌 클래스를 상속받아서 메소드를 오버라이딩 할 때는 대부분 상위 클래스의 메소드를 호출하고 새로운 내용을 추가
- 프레임워크에서 제공하는 클래스의 메소드 들은 본연의 기능이 있으므로 그 기능을 수행하고 다른 기능을 추가해야 하는 경우가 있습니다.
- Overriding을 이용해서 Polymorphism(다형성)을 구현할 수 있는 있는데 Polymorphism 이란 동일한 메시지에 대하여 다르게 반응하는 성질을 의미하는데 오버라이딩이나 함수 포인터를 이용해서 구현합니다.

상속 – Overriding(재정의)

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

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

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

A().method()
B().method()
C().method()
```

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

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

상속 – Overriding(재정의)

```
class Star:
    def attack(self):
        print("스타의 어택")

class Terran(Star):
    def attack(self):
        print("테란의 어택")

class Zerg(Star):
    def attack(self):
        print("저그의 어택")

obj = Terran();
obj.attack()
obj = Zerg()
obj.attack()
```

obj.attack()이라는 코드가 두번 사용되었는데 어떤 객체를 생성했느냐에 따라 호출되는 메소드가 달라집니다.

상속 – 다중 상속

- 다중상속은 자식 하나가 여러 부모(?!)로부터 상속을 받는 것
  - 파생 클래스의 정의에 기반 클래스의 이름을 콤마(,)로 구분해서 쭉 적어주면 다중상속이 이루어짐.

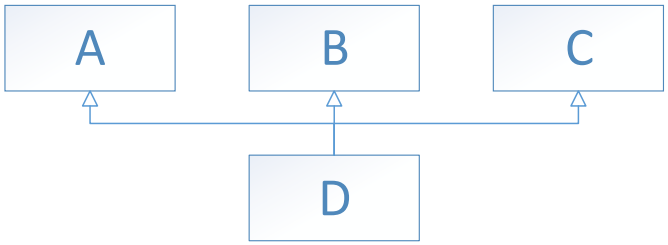
```
class A:
    pass

class B:
    pass

class C:
    pass

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

클래스 D는 클래스 A, B, C 클래스로부터 상속받습니다.



상속 – 다중 상속

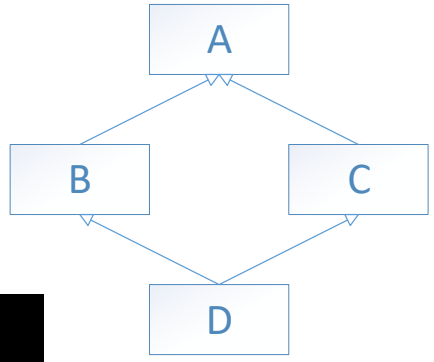
- 다이아몬드 상속 : 다중 상속이 만들어 내는 곤란한 상황
  - D는 B와 C 중 누구의 method()를 물려받게 되는 걸까?

```
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.

위임(Delegation)

- 어떤 객체가 자신이 처리할 수 없는 메시지를 받으면 해당 메시지를 처리할 수 있는 다른 객체에 전달하는 기법
- __getattr__()를 구현하면 되는데 이 메소드의 매개변수는 name 매개변수를 추가로 갖습니다.
- name은 참조하는 속성 이름이 넘어가게 됩니다.

```
class Delegation:
    def __init__(self, li):
        self.li = li

    def __getattr__(self, name):
        print("Delegation {0}".format(name), end=' ')
        return getattr(self.li, name)

ob = Delegation([1,3,5,7,9,1])
print(ob.pop())
print(ob.count(3))
```

Delegation 클래스에는 pop 메소드가 없어서 __getattr__ 메소드를 호출합니다.

데코레이터

- **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

```
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.")

print_hello = MyDecorator(print_hello)

print_hello()
```

MyDecorator의 func 데이터 속성이 print_hello를 받아둡니다.

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

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

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

데코레이터

```
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()
```

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

이터레이터

- 파이썬에서 for문을 실행할 때 가장먼저 하는 일은 순회하려는 객체의 __iter__() 메소드를 호출하는 것.
  - __iter__() 메소드는 이터레이터(Iterator)라고 하는 특별한 객체를 for문에게 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.

```
>>> 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
```

이터레이터

- 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)
```

```
0
1
2
3
4
```

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 중단시켜 놓기만 함.

```
def YourRange(start, end):
    current = start
    while current < end:
        yield current
        current += 1
    return

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

```
0
1
2
3
4
```

type()

- 메타 클래스(Meta **Class**)
  - 클래스를 만들어내는 클래스
  - 파이썬은 클래스도 하나의 객체로 취급해서 변수에 치환을 할 수 있고 복사도 가능하고 속성 값을 동적으로 추가할 수 있고 함수의 매개변수로 대입할 수 있습니다.
  - 객체이므로 다른 코드 블록 내에서 생성이 가능
- type() 함수
  - 매개변수로 객체를 대입하면 객체의 자료형을 리턴합니다.
  - 다른 용도로는 클래스의 기능을 동적으로 확장하는 역할을 합니다.
  - 클래스를 만들 때는 첫번째 매개변수는 클래스이름이고 두번째 매개변수는 상위 클래스 이름이고 세번째 매개변수는 속성과 값을 디셔너리로 대입하면 됩니다.

type()

- 클래스 기능 확장

```
class StaticClass:
    attr = "정적 클래스"

ob1 = StaticClass()
print(ob1.attr)
ob2 = type('DynamicClass', (), {'attr':'동적 클래스'})
print(ob2.attr)
```

추상 클래스

- 추상 클래스(Abstract **Class**)
  - 하위 클래스들의 특징을 소유하고 있는 클래스
  - 추상 클래스를 정의할 때는 abc 모듈의 ABCMeta 클래스를 상속받아야 합니다.
  - Metaclass를 상속받을 때는 **metaclass=메타클래스이름**
  - 추상 메소드는 이름만 존재하고 내용이 없는 메소드
  - 추상 메소드는 @abstractmethod 데코레이터를 이용
  - 추상 클래스는 자신의 객체를 생성할 수 없습니다.
  - 상속을 받아서 하위 클래스에서 메소드를 재정의해서 사용합니다.

추상 클래스

- 추상 클래스

```
from abc import *
class AbstractClass(metaclass = ABCMeta):
    attr = "추상 클래스"
    @abstractmethod
    def abstractMethod(self):
        pass

ob = AbstractClass()
```

추상 클래스를 객체화 시킬려고 하므로 에러

```
Traceback (most recent call last):
  File "C:WUsersWAdministratorWpythonWtestW__main__.py", line 8, in <module>
    ob = AbstractClass()
TypeError: Can't instantiate abstract class AbstractClass with abstract methods abstractMethod
```

싱글톤(Singleton)

- **Singleton Class**
  - 객체를 1개만 만들 수 있는 클래스
  - 공유 데이터를 소유하는 클래스나 관리자 클래스 또는 서버 측에서 클라이언트의 요청을 처리하는 클래스를 Singleton으로 만드는 경우가 많습니다.

```
class Singleton(type):
    __instances = {}
    def __call__(self, *args, **kwargs):
        if self not in self.__instances:
            self.__instances[self] = super().__call__(*args, **kwargs)
        return self.__instances[self]

class MyClass(metaclass=Singleton):
    pass

m1 = MyClass()
m2 = MyClass()
print(id(m1))
print(id(m2))
```

싱글톤 클래스이므로 2개 객체의 id가 동일하게 나옵니다.

final

- **final Class**
  - 상속을 하지 못하는 클래스

```
class final(type):
    def __init__(cls, name, bases, namespace):
        super().__init__(name, bases, namespace)
        for klass in bases:    # 기반 클래스에 final이 있으면 에러를 발생시킨다
            if isinstance(klass, final):
                raise TypeError(klass.__name__ + ' is final')


class A: pass
class B(A, metaclass=final): pass
class C(B): pass
```

B는 final 클래스이므로 상속할 수 없다.