클래스

Class

- 사용자 정의 데이터 타입 : 데이터 + 함수

```
class 클래스명 :
    클래스 변수
    클래스 함수(self, 인수)
```

인스턴스 객체

```
class Service:
    secret = "영구는 배꼽이 두 개다"
    def sum(self, a, b):
        result = a + b
        print("%s + %s = %s 입니다." % (a, b, result))

pey = Service()
print(pey.secret)

pey.sum(1,1)  # bound method call
Service.sum(pey, 1,1) # unbound method call
```

Class

- 생성자 : __init__ 함수
  - 객체를 만들때 호출되는 함수
  - 속성 초기화 할때 사용, 속성은 공유되지 않음

```
class Service:
    secret = "영구는 배꼽이 두 개다"      # 클래스 변수
    def __init__(self, name, value=0):
        self.name = name          # 인스턴스 변수
        self.value = value
        self.secret = name
    def sum(self, a, b):
        result = a + b
        print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))

Service.age=0  # 새로운 클래스 변수 추가
S1.age2 = 20   # S1 인스턴스에서만 사용할 수 있는 새로운 멤버
```

Class

- 인스턴스 객체의 내장 속성 __class__
  - 클래스 멤버를 액세스 하기 위한 속성

```
class Service:
    secret = "영구는 배꼽이 두 개다"      # 클래스 변수
    def __init__(self, name, value=0):
        self.name = name          # 인스턴스 변수
        self.value = value
        self.secret = name
    def sum(self, a, b):
        result = a + b
        print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))
```

s1.secret =“영구는 배꼽이 세개다”
s1.__class__.secret = “영구는 배꼽이 네개다”

Class

- 소멸자 : __del__ 함수
  - 객체가 소멸될 때 자동으로 호출되는 함수
  - 인스턴스 객체의 참조 카운터가 0이 될 때 호출
    - **del 함수로 삭제 가능 (del 객체)**

```
class HousePark:
    lastname = "박"

    def __init__(self, name):
        self.fullname = self.lastname + name

    def __del__(self):
        print(self.fullname +" 객체가 소멸합니다.")

    def travel(self, where):
        print("%s, %s여행을 가다." % (self.fullname, where))
```

class

- Static Method
  - 인스턴스 객체를 통하지 않고 클래스를 통해 직접 호출 가능한 메소드
  - 메소드 정의시 인스턴스를 참조하는 self라는 인자를 선언하지 않음

  # 정적 메소드 정의
  메소드 정의

  # 정적 메소드 등록
  호출할 메소드 이름 = staticmethod(클래스내 정의한 메소드 이름)

  Or

  @staticmethod #데코레이터
  메소드 정의

class

- Class Method
  - 암묵적으로 클래스 객체를 인자로 전달받으므로, 클래스의 변수를 사용할 수 있음

클래스 메소드 정의
메소드 정의

클래스 메소드 등록
호출할 메소드 이름 = classmethod(클래스내 정의한 메소드 이름)

Or

@classmethod #데코레이터
메소드 정의

class

```
class MyClass:
    data=1

    def classTest(cls):
        print("class method")
        print(cls.data)
        print()
    CTest = classmethod(classTest)

    def staticTest():
        print("static method")
        print()
    STest = staticmethod(staticTest)

MyClass.CTest()
MyClass.STest()
```

Class 상속

class 클래스명(상속 클래스명) :

클래스 변수

클래스 함수(**self**, 인수)

*모든 클래스는 object로 부터 파생됨

```
class HousePark:
    lastname = "박"
    def __init__(self, name):
        self.fullname = self.lastname + name
    def travel(self, where):
        print("%s, %s여행을 가다." % (self.fullname, where))

class HouseKim(HousePark):
    lastname = "김"
    def __init__(self, name, age):
        HousePark.__init__(self, name)
        self.age = age
    def travel(self, where, day):
        HousePark.travel(self, where)
        print("%s, %d살에 %s여행 %d일 가네." % (self.fullname, self.age, where, day))
```

Class 다중 상속

class 클래스명(상속 클래스명1, 상속 클래스명2) :

클래스 변수

클래스 함수(**self**, 인수)

```
class A :
    def __init__(self):
        print("A 생성자 호출")
class B :
    def __init__(self):
        print("B 생성자 호출")
class C(A,B) :
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print("C 생성자 호출")
```

Class 다중 상속

- Super()
  - 부모 클래스의 객체를 반환
    - issubclass(SubClass, SuperClass)
    - SubClass.__bases__ : super class 정보 출력

```
class A :
    def __init__(self):
        print("A 생성자 호출")
class B(A) :
    def __init__(self):
        A.__init__(self)
        print("B 생성자 호출")
class C(A) :
    def __init__(self):
        A.__init__(self)
        print("C 생성자 호출")
class D(B,C):
    def __init__(self):
        B.__init__(self)
        C.__init__(self)
        print("D 생성자 호출")
```

```
class A :
    def __init__(self):
        print("A 생성자 호출")
class B(A) :
    def __init__(self):
        super().__init__()
        print("B 생성자 호출")
class C(A) :
    def __init__(self):
        super().__init__()
        print("C 생성자 호출")
class D(B,C):
    def __init__(self):
        super().__init__()
        print("D 생성자 호출")
```

Class 다중 상속

```
class A():
    def __init__(self, a):
        self.a = a
    def show(self):
        print('show:', self.a)

class B(A):
    def __init__(self, b, **arg):
        super().__init__(**arg)
        self.b = b
    def show(self):
        print('show:', self.b)
        super().show()

class C(A):
    def __init__(self, c, **arg):
        super().__init__(**arg)
        self.c = c
    def show(self):
        print('show:', self.c)
        super().show()
```

```
class D(B,C):
    def __init__(self, d, **arg):
        super().__init__(**arg)
        self.d = d
    def show(self):
        print('show:', self.d)
        super().show()

data = D(a=1,b=2,c=3,d=4)
data.show()
```

*오버라이딩한 함수 접근하기
super(Subclass, self).함수이름()
  Ex) super(C, data).show()

Private Variables

- __ 을 사용하여 private variables 란 개념으로 지원함
  - __ 가 붙은 멤버는 모두 앞에 현재 class 정보 __class__가 추가 되어 실힘됨
  - 클래스 밖에서 실행할 경우 클래스가 다르기 때문에 액세스 할 수 없게 함

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

# private copy of original update() method
    __update = update

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

추상 Class

- 추상메소드를 포함하고 있는 클래스
  - 자식클래스는 반드시 추상메소드를 구현해야 함
    - 추상메소드 : 자식클래스에서 꼭 재정의 되어야 하는 메소드

```
from abc import ABCMeta, abstractmethod

class 클래스이름(metaclass=ABCMeta):
    @abstractmethod
    def 추상메소드(self):
        pass
```