기초빅데이터프로그래밍

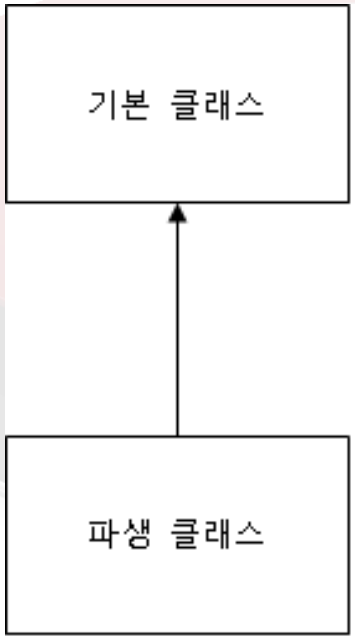
객체지향 프로그래밍 Part3

서강대학교
SOGANG UNIVERSITY

8 상속

- 부모 클래스의 속성(데이터, 메서드)을 자식 클래스로 물려줄 수 있다.
- 상속을 하는 이유
  - 기존의 설계를 바탕으로 추가된 부분만 만들기 때문에 빠르게 만들 수 있다. 이를 재사용(reuse)이라 하며, 생산성과도 관련이 있다.
  - 기존의 설계가 안정적이면 이를 바탕으로 만드는 것이 안정성을 보장 받을 수 있다.
- 상속의 문법

```
class 클래스_이름(기본_클래스):
    클래스_본체
```



```
|: class Parent:
       def can_sing(self):
           print("Sing a song")

   father = Parent()
   father.can_sing()
```

```
Sing a song
```

```
|: class LuckyChild(Parent):
       pass

   child1 = LuckyChild()
   child1.can_sing()
```

```
Sing a song
```

상속받은 child1는 노래할 수 있지만,
상속받지 않은 child2는 노래 못한다.

```
|: class UnLuckyChild:
       pass

   child2 = UnLuckyChild()
   child2.can_sing()
```

```
---------------------------------------
---
AttributeError
st)
<ipython-input-4-aacd6c77e90b> in <modul
      3
      4 child2 = UnLuckyChild()
----> 5 child2.can_sing()

AttributeError: 'UnLuckyChild' object has no attribute 'can_sing'
```

```
In [7]: class LuckyChild2(Parent):
            def can_dance(self):
                print("Shuffle Dance")

        child2 = LuckyChild2()
        child2.can_sing()
```

```
Sing a song
```

```
In [8]: child2.can_dance()
```

```
Shuffle Dance
```

sportcar_example1.py

```
class SportCar(object):
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 40

    def turbocharge(self):
        self._speed = self._speed + 70

    def stop(self):
        self._speed = 0
```

```
if __name__ == "__main__":
    my_sportcar = SportCar()
    my_sportcar.start()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.accelerate()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.turbocharge()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.stop()
```

상속받지 않고 직접설계

```
class SportCar(object):
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 40

    def turbocharge(self):
        self._speed = self._speed + 70

    def stop(self):
        self._speed = 0

if __name__ == "__main__":
    my_sportcar = SportCar()
    my_sportcar.start()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.accelerate()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.turbocharge()
    print("속도:", my_sportcar.get_speed())
    my_sportcar.stop()
```

```
속도: 20
속도: 60
속도: 130
```

```
# sportcar_example2.py
#car를 상속받아 sportCar설계
class Car(object):
    def __init__(self):
        self._speed = 0

    @property
    def speed(self):
        return self._speed

    def start(self):
        self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0
```

```
class SportCar(Car):
    def __init__(self):
        self._color = "red"

    def accelerate(self):
        self._speed = self._speed + 40

    def turbocharge(self):
        self._speed = self._speed + 70

    @property
    def color(self):
        return self._color


if __name__ == "__main__":
    my_sportcar = SportCar()
    print("색상:", my_sportcar.color)
    my_sportcar.start()
    print("속도:", my_sportcar.speed)
    my_sportcar.accelerate()
    print("속도:", my_sportcar.speed)
    my_sportcar.turbocharge()
    print("속도:", my_sportcar.speed)
    my_sportcar.stop()
```

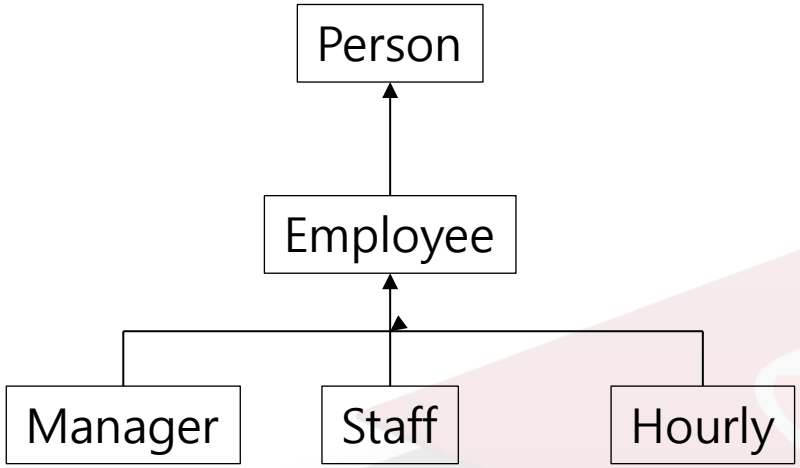
상속받아 설계

```
class Car(object):
    def __init__(self):
        self._speed = 0
    @property
    def speed(self):
        return self._speed
    def start(self):
        self._speed = 20
    def accelerate(self):
        self._speed = self._speed + 30
    def stop(self):
        self._speed = 0

class SportCar(Car):
    def __init__(self):
        self._color = "red"
    @property
    def color(self):
        return self._color
    def accelerate(self):
        self._speed = self._speed + 40
    def turbocharge(self):
        self._speed = self._speed + 70
```

```
if __name__ == "__main__":
    my_sportcar = SportCar()
    print("색상:", my_sportcar.color)
    my_sportcar.start()
    print("속도:", my_sportcar.speed)
    my_sportcar.accelerate()
    print("속도:", my_sportcar.speed)
    my_sportcar.turbocharge()
    print("속도:", my_sportcar.speed)
    my_sportcar.stop()
```

```
색상: red
속도: 20
속도: 60
속도: 130
```

실습



Person, Employee, Manager, Staff, Hourly class를 구성해서 작은 회사를 만들어 보세요.

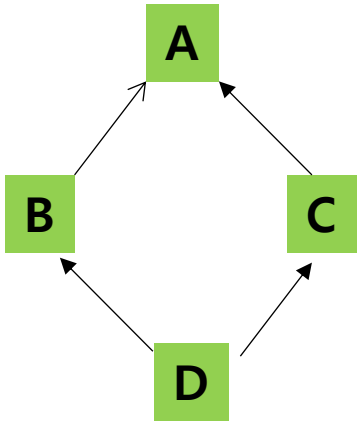
```
class Person:
    def __init__(self, name, age, gender):
        self.Name = name
        self.Age = age
        self.Gender = gender
    def aboutMe(self):
        print("이름은 " + self.Name + "이고, 나이는 " + self.Age + "살 입니다.")

class Employee(Person):
    def __init__(self, name, age, gender, salary, hiredate):
        Person.__init__(self, name, age, gender)
        self.Salary = salary
        self.Hiredate = hiredate
    def doWork(self):
        print("열심히 일합니다.")
    def aboutMe(self):
        Person.aboutMe(self)
        print("급여는 " + self.Salary + "원이고, 입사일은 " + self.Hiredate +"입니다" )

employee1 = Employee("김철수", "18", "남", "3000000", "2016년 3월 1일")
employee1.doWork()
```

```
열심히 일합니다.
```

```
employee1.aboutMe()
```

```
이름은 김철수이고, 나이는 18살 입니다.
급여는 3000000원이고, 입사일은 2016년 3월 1일입니다
```

다중상속




```
class A:
    def __init__(self):
        print("A class의 생성자 호출")

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

class C(A):
    def __init__(self):
        A.__init__(self)
        print("C class의 생성자 호출")

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

ObjectD = D()
```

```
A class의 생성자 호출
B class의 생성자 호출
A class의 생성자 호출
C class의 생성자 호출
D class의 생성자 호출
```

```
class A:
    def __init__(self):
        print("A class의 생성자 호출")

class B(A):
    def __init__(self):
        super().__init__()
        print("B class의 생성자 호출")

class C(A):
    def __init__(self):
        super().__init__()
        print("C class의 생성자 호출")

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

ObjectD = D()
```

```
A class의 생성자 호출
C class의 생성자 호출
B class의 생성자 호출
D class의 생성자 호출
```

super() 함수는 슈퍼클래스의 method 를 호출하라는 의미인데, 이때 다수의 슈퍼 클래스가 존재 시 클래스 호출 순서의 결정은 __mro__ 를 통해 결정 된다.

mro (Method Resolution Order)

```
class A:
    def __init__(self):
        print("Class A __init__()")

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

class C(A):
    def __init__(self):
        print("Class C __init__()")
        super(C, self).__init__()

class D(B,C):
    def __init__(self):
        print("Class D __init__()")
        super(D, self).__init__()

d = D()
```

```
Class D __init__()
Class B __init__()
Class C __init__()
Class A __init__()
```

```
D.__mro__
```

```
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

```
C.__mro__
```

```
(__main__.C, __main__.A, object)
```

```
B.__mro__
```

```
(__main__.B, __main__.A, object)
```

```
A.__mro__
```

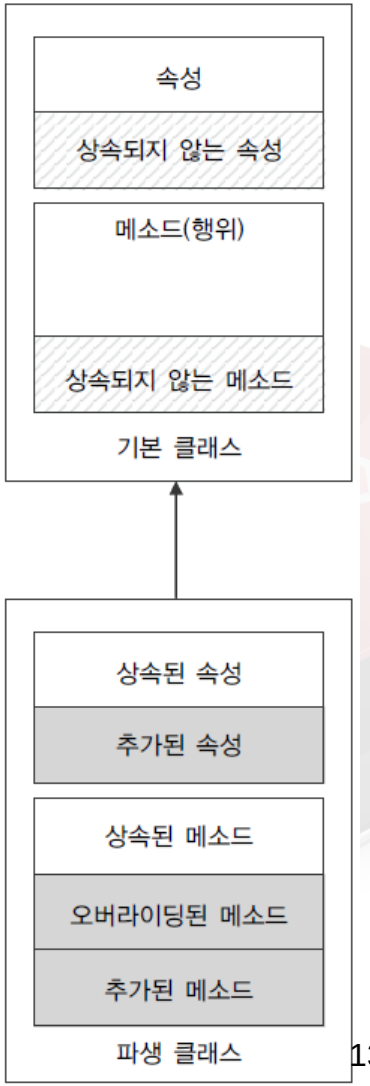
```
(__main__.A, object)
```

python 3.X에서는 문제 없이 동작하지만, python 2.X 에서는 에러가 발생한다.

super(type[, object-or-type]), 첫째 argument는 반드시 *type* 형태의 new-style class 이어야 한다.
python 2.X 에서는 class를
class A(object):
으로 정의해야 old-style class가 아닌 new-style class로 생성된다.
old-style class는 type이 아닌 classobj이므로...

9. 추가와 오버라이딩(Overriding, 재정의)

- 기본 클래스에 없는 속성과 메서드를 파생 클래스에 추가한다.
- 상속받은 메서드의 동작이 적합하지 않은 경우에는 파생 클래스 쪽에서 이를 재정의(overriding)한다.
- 기본 클래스에서 상속을 하지 않아야 할 속성이나 행위는 **밑줄을 두 개 붙여서 상속을 막는다.**



10 다형성(Polymorphism)

- 다형성이란 각 객체가 동일한 인터페이스에 대해 서로 다른 동작을 수행하도록 하는 것을 말한다.
  - 객체가 다형성을 띄면 객체 사용에 있어 편리성이 증대된다.
- 객체지향에서 말하는 **인터페이스는 구현부가 없는 메서드이다**.
- 실질적인 구현은 상속받은 클래스에 있다.

같은 이름의 메서드의 내부 로직을 다르게 정의
getArea()의 계산방법이 다름

In [1]:
```
class Figure:
    def __init__(self, width,height):
        self.width = width
        self.height = height
    def getArea(self):
        pass

class Triangle(Figure):
    def getArea(self):
        return self.width * self.height/2.0

class Rectangle(Figure):
    def getArea(self):
        return self.width * self.height

myTriangle = Triangle(10, 10)
myRectangle = Rectangle(10,10)

print("Triangle Area is ", myTriangle.getArea())
print("Rectangle Area is ", myRectangle.getArea())
```

```
Triangle Area is  50.0
Rectangle Area is  100
```

AirForce 인터페이스를 정의

```
# airforce.py

class AirForce(object):  #AirForce(공군기)의 인터페이스 정의
    def take_off(self):
        pass

    def fly(self):
        pass

    def attack(self):
        pass

    def land(self):
        pass
```

AirForce 인터페이스를 상속 받은 Fighter 클래스 정의

서강대학교 SOGANG UNIVERSITY

```
# fighter.py
from airforce import AirForce

class Fighter(AirForce): # 전투기 클래스
    def __init__(self, weapon_num):
        self._missile_num = weapon_num

    def take_off(self):
        print("전투기 발진")

    def fly(self):
        print("전투기가 목표지로 출격")

    def attack(self):
        for i in range(self._missile_num):
            print("미사일 발사")
            self._missile_num = self._missile_num – 1

    def land(self):
        print("전투기 착륙")
```

AirForce 인터페이스를 상속 받은 Bomber 클래스 정의

서강대학교 SOGANG UNIVERSITY

 # bomber.py

```
from airforce import AirForce

class Bomber(AirForce): # 1 # 폭격기 클래스
    def __init__(self, bomb_num):
        self._bomb_num = bomb_num

    def take_off(self):
        print("폭격기 발진")

    def fly(self):
        print("폭격기 목적지로 출격")

    def attack(self):
        for i in range(self._bomb_num):
            print("폭탄 투하")
            self._bomb_num = self._bomb_num - 1

    def land(self):
        print("폭격기 착륙")
```

```
# war_game.py
from fighter import Fighter
from bomber import Bomber

def war_game(airforce):
    airforce.take_off()
    airforce.fly()
    airforce.attack()
    airforce.land()

if __name__ == "__main__":
    f15 = Fighter(3)
    war_game(f15)
    print()
    b29 = Bomber(3)
    war_game(b29)
```

```
from fighter import Fighter
from bomber import Bomber

def war_game(airforce):
    airforce.take_off()
    airforce.fly()
    airforce.attack()
    airforce.land()

if __name__ == "__main__":
    f15 = Fighter(3)
    war_game(f15)
    print()
    b29 = Bomber(3)
    war_game(b29)
```

```
전투기 발진
전투기가 목표지로 출격
미사일 발사
미사일 발사
미사일 발사
전투기 착륙

폭격기 발진
폭격기 목적지로 출격
폭탄 투하
폭탄 투하
폭탄 투하
폭격기 착륙
```

11 인스턴스 속성과 클래스 속성

- 인스턴스 속성
  - 객체가 가지고 있는 속성

- 클래스 속성
  - 클래스가 가지고 있는 속성
  - 클래스에 하나 존재하고 있으며 이는 모든 객체에서 공유한다.

- 클래스 속성은 클래스 레벨(객체의 메서드 외부)에 정의된다.
  - 클래스 속성을 접근할 때는 **클래스 이름을 이용해서 접근한다**.

```
# class_variable.py

class Circle(object):
    PI = 3.14 # 클래스 변수

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    # 원의 면적을 구한다
    def get_area(self): # 클래스 변수 PI이용
        area = Circle.PI * (self._radius ** 2)
        return round(area, 2)

    # 원의 둘레를 구한다.
    def get_circumference(self): # 3
        circumference = 2 * self.PI * self._radius
        return round(circumference, 2)
```

```
if __name__ == "__main__":
    circle1 = Circle(3)
    print("원주율: ", Circle.PI) # 4
    print("반지름: ", circle1.radius, "면적: ", circle1.get_area())
    print("반지름: ", circle1.radius, "둘레: ",
    circle1.get_circumference())
    circle2 = Circle(4)
    print("반지름: ", circle2.radius, "면적: ", circle2.get_area())
    print("반지름: ", circle2.radius, "둘레: ",
    circle2.get_circumference())
```

#3 클래스 변수 이용에 self.PI도 가능하지만, 클래스이름을 이용하는 것이 좋다.

```
class Circle(object):
    PI = 3.14 # 클래스 변수

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    # 원의 면적을 구한다
    def get_area(self): # 클래스 변수 PI이용
        area = Circle.PI * (self._radius ** 2)
        return round(area, 2)

    # 원의 둘레를 구한다.
    def get_circumference(self): # 3
        circumference = 2 * self.PI * self._radius
        return round(circumference, 2)
if __name__ == "__main__":
    circle1 = Circle(3)
    print("원주율: ", Circle.PI) # 4
    print("반지름: ", circle1.radius, "면적: ", circle1.get_area())
    print("반지름: ", circle1.radius, "둘레: ",
    circle1.get_circumference())
    circle2 = Circle(4)
    print("반지름: ", circle2.radius, "면적: ", circle2.get_area())
    print("반지름: ", circle2.radius, "둘레: ",
    circle2.get_circumference())
```

```
원주율:  3.14
반지름:  3 면적:  28.26
반지름:  3 둘레:  18.84
반지름:  4 면적:  50.24
반지름:  4 둘레:  25.12
```

12 인스턴스 메서드, 클래스 메서드, 그리고 정적 메서드

- **세가지 모두 클래스 안에서 정의됨**
- **인스턴스(Instance) 메서드**
  - **객체(인스턴스)를 통해서** 사용되는 메서드
  - 객체를 생성한 이후에 사용이 가능하며 주로 객체의 속성을 조작, 관리 및 정보의 생성에 이용된다.
  - 인스턴스 변수에 엑세스할 수 있도록 메서드의 첫번째 파라미터에 항상 객체 자신을 의미하는 "**self**"라는 파라미터를 갖는다. 즉, 첫 번째 인자 self로 인스턴스 자신이 자동으로 호출된다.

Class method

- **클래스(class) 메서드**
  - **클래스를 통해서** 사용하며, 따라서 객체의 생성 없이도 사용이 가능하다. (객체와 무관하므로 self가 없음에 유의)
  - 클래스 메서드에서 객체의 속성을 접근하거나 사용하려는 행위는 오류를 발생시킨다. (객체와 관련이 없다)
  - **cls** 라는 클래스를 의미하는 파라미터를 전달받아, 클래스 변수 등을 액세스 할 수 있다. 즉, 첫 번째 인자로는 클래스 자신이 자동으로 전달되고 이 인수를 '**cls**'라고 한다
  - 메서드 앞에 **@classmethod** 라는 Decorator를 표시

X = ?

```
class Times(object):

        factor = 1
        @classmethod
        def mul(cls, x):
                return cls.factor * x

class TwoTimes(Times):
        factor =2


x = TwoTimes.mul(4)
```

classmethod_example.py

```
class CircleCalculator(object):
    __PI = 3.14   #클래스 속성, 외부에서 접근불가

    # 원의 면적을 구하는 클래스 메서드
    @classmethod   #클래스 메서드는 반드시 @classmethod로 데코레이트 되어야 함
    def calculate_area(cls, radius):
        area = cls.__PI * (radius ** 2)
        return round(area, 2)

    # 원의 둘레를 구하는 클래스 메서드
    @classmethod
    def calculate_circumference(cls, radius):
        circumference = 2 * cls.__PI * radius
        return round(circumference, 2)

if __name__ == "__main__":
    print("반지름:", 3, "면적:",CircleCalculator.calculate_area(3))
    print("반지름:", 3, "둘레:",CircleCalculator.calculate_circumference(3))
```

클래스 메서드의 첫번째 인자는 cls이며, 이 경우 CircleCalculator를 가리킨다. 클래스메서드 내부에서는 cls를 이용해서 클래스속성에 접근한다.

객체를 생성하지 않고, 클래스 메서드를 사용해서 원의 면적과 둘레를 구하기

```
class CircleCalculator(object):
    __PI = 3.14   #클래스 속성, 외부에서 접근불가

    # 원의 면적을 구하는 클래스 메서드
    @classmethod   #클래스 메서드는 반드시 @classmethod로 데코레이트 되어야 함
    def calculate_area(cls, radius):
        area = cls.__PI * (radius ** 2)
        return round(area, 2)

    # 원의 둘레를 구하는 클래스 메서드
    @classmethod
    def calculate_circumference(cls, radius):
        circumference = 2 * cls.__PI * radius
        return round(circumference, 2)

if __name__ == "__main__":
    print("반지름:", 3, "면적:",CircleCalculator.calculate_area(3))
    print("반지름:", 3, "둘레:",CircleCalculator.calculate_circumference(3))
```

```
반지름: 3 면적: 28.26
반지름: 3 둘레: 18.84
```

Static method

- **정적(static) 메서드**
  - 클래스에 소속된 함수
  - 클래스 메서드처럼 객체 생성 없이 클래스를 통해서 사용이 가능하다.
  - **정적 메서드**는 인스턴스 메서드에서와 같은 self 파라미터를 갖지 않고 인스턴스 변수에 엑세스할 수 없다.
  - 정적 메서드는 보통 **객체 필드와 독립적이지만** 로직상 클래스내에 포함되는 메서드에 사용된다.
  - 클래스 객체로 생성된 모든 인스턴스 객체가 공유하여 사용할 수 있다.
  - 정적 메서드와 클래스 메서드의 차이점은 클래스 메서드의 경우 첫 번째 인자로 cls를 넘겨받지만 정적 메서드는 첫 번째 인자로 cls를 넘겨받지 않는다.
  - 메서드 앞에 **@staticmethod** 라는 Decorator를 표시한다

앞의 classmethod_example.py의 class method를 단순히 static method로 바꾼 경우

```
class CircleCalculator(object):
    __PI = 3.14   #클래스 속성, 외부에서 접근불가

    @staticmethod
    def calculate_area(radius):
        area = CircleCalculator.__PI * (radius ** 2)
        return round(area, 2)

    @staticmethod
    def calculate_circumference(radius):
        circumference = 2 * CircleCalculator.__PI * radius
        return round(circumference, 2)

if __name__ == "__main__":
    print("반지름:", 3, "면적:",CircleCalculator.calculate_area(3))
    print("반지름:", 3, "둘레:",CircleCalculator.calculate_circumference(3))
```

```
반지름: 3 면적: 28.26
반지름: 3 둘레: 18.84
```

- ✓ 정적메서드로 선언하고 클래스변수를 이용해도 결과는 나오지만, 다음 페이지처럼 매개변수로 pi를 전달 받는 것이 좋다.
- ✓ 클래스 변수를 액세스할 필요가 없을 경우에 정적메서드를 사용하는 것이 좋다.

```
# staticmethod_example.py

class CircleCalculator(object):
    # 원의 면적을 구하는 정적 메서드
    @staticmethod
    def calculate_area(radius, pi):
        area = pi * (radius ** 2)
        return round(area, 2)

    # 원의 둘레를 구하는 정적 메서드
    @staticmethod
    def calculate_circumference(radius, pi):
        circumference = 2 * pi * radius
        return round(circumference, 2)

if __name__ == "__main__":
    print("반지름:", 3, "면적:",CircleCalculator.calculate_area(3, 3.14))
    print("반지름:", 3, "둘레:",CircleCalculator.calculate_circumference(3, 3.14))
```

정적 메서드는 반드시 @staticmethod로 데코레이트 되어야함

```python
import time

class Date(object):

    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @staticmethod
    def now():
        t = time.localtime()
        return Date(t.tm_year, t.tm_mon, t.tm_mday)

# example

a = Date(2011,8,14)
print('{}년{}월 {}일 입니다.'.format(a.year, a.month, a.day))
b = Date.now()
print('{}년{}월 {}일 입니다.'.format(b.year, b.month, b.day))
```

```
2011년8월 14일 입니다.
2018년4월 1일 입니다.
```

Static method는 일종의 utility task들을 한쪽에 몰아넣고 사용하거나

```
class UTIL:
    @staticmethod
    def addtask(a,b):
        return a+b
x = UTIL.addtask(1,4)  # x = 5
```

또는, init 구문이 복수 개로 들어가는 경우, 일종의 편법으로 생성을 해서 자기자신을 리턴하는 형태로 사용한다.

실습

- static method를 이용해서 다음과 같이 수행되도록 Date 클래스를 만드시오.

```
a = Date("2022, 4, 7")
a.show()
b = Date.now()
b.show()
c = Date.yesterday("2022, 4, 7")
c.show()
```

```
date: 2022, 4, 7
date: today
date: 2022, 4, 6
```

```
a = Date("2022, 4, 7")
a.show()
b = Date.now()
print(b.date)
c = Date.yesterday("2022, 4, 7")
print(c.date)
```

```
date: 2022, 4, 7
date: today
date: 2022, 4, 6
```

인스턴스 데이타를 엑세스할 필요가 없는 경우 클래스 메서드나 정적 메서드를 사용하는데, 이때 **클래스 변수를 엑세스할 필요가 없을 때는 정적 메서드를 사용한다.**

- isSquare() 메서드는 cls 파라미터없고, 메서드 내에서 클래스 변수를 사용하지 않고 있다.
- printCount() 메서드는 cls 파라미터를 전달받고 메서드 내에서 클래스 변수 count를 사용하고 있다.

```
class Rectangle:
    count = 0  # 클래스 변수

    def __init__(self, width, height):
        self.width = width
        self.height = height
        Rectangle.count += 1

    def calcArea(self):  # 인스턴스 메서드
        area = self.width * self.height
        return area

    @staticmethod   # 정적 메서드
    def isSquare(rectWidth, rectHeight):
        return rectWidth == rectHeight

    @classmethod
    def printCount(cls):
        print(cls.count)

square = Rectangle.isSquare(5, 5)
print(square)   # True

rect1 = Rectangle(5, 5)
rect2 = Rectangle(2, 5)
rect1.printCount()  # 2
```

```
True
2
```

클래스 변수를 엑세스할 때, "**클래스명.클래스변수명**" 혹은 "**객체명.클래스변수명**"을 둘 다 허용하지만... 혼돈을 피하기 위해 클래스 변수를 엑세스할 때는 클래스명을 사용하는 것이 좋다.

In [2]:
```
# 객체 생성
r = Rectangle(2, 3)

# 메서드 호출
area = r.calcArea()
print("area = ", area)

# 인스턴스 변수 엑세스
r.width = 10
print("width = ", r.width)

# 클래스 변수 엑세스
print(Rectangle.count)
print(r.count)
```

```
area =  6
width =  10
3
3
```

값이 같은 이유: r 객체에서 count를 읽기만 하므로 새 인스턴스 변수를 생성하지 않았으므로 객체의 attribute가 없어서 클래스의 attribute를 찾았기 때문이다.

클래스 변수인 count를 사용하는 것이 아니라 새로 그 객체에 추가된 인스턴스 변수를 사용하게 되므로 클래스 변수값은 변경되지 않는다.

```
r = Rectangle(2, 4)

Rectangle.count = 50
r.count = 10    # count 인스턴스 변수가 새로 생성됨

print(r.count, Rectangle.count)  # 10  50 출력
```

```
10 50
```

정수 클래스를 상속받은 MyInt클래스를 만들어 덧셈을 수정해보자

```
class MyInt(int):
    pass

my_num = MyInt(5) # 인스턴스 생성

my_num + 5
```

```
10
```

```
my_num.__add__(5)
```

```
10
```

```
class MyInt(int):

    def __add__(self, other): # __add__ 변경
        return '{} 더하기 {} 는 {} 입니다'.format(self.real, other.real, self.real + other.real)

my_num = MyInt(6)

print(my_num + 5)  # => 6 더하기 5 는 11 입니다
```

```
6 더하기 5 는 11 입니다
```

```
my_num2=MyInt(8)
my_num + my_num2
```

```
'6 더하기 8 는 14 입니다'
```

```
6 + 8
```

```
14
```

실습

- 아래와 같은 Food class의 객체들의 가격을 비교해서 크기를 결정하도록 __lt__ method를 수정하시오.

```
class Food(object):
    def __init__(self, name, price):
        self.name = name
        self.price = price

food_1 = Food('아이스크림', 3000)
food_2 = Food('햄버거', 5000)

# food_1이 food_2보다 작은지 확인
print(food_1)
print(food_2)
print(food_1 < food_2)  #파이썬2에서는 주소를 비교해서 FALSE를 return해줌
```

```
<__main__.Food object at 0x000002F2402
<__main__.Food object at 0x000002F2402

---------------------------------------
TypeError
<ipython-input-5-05e95a049714> in <mod
     10 print(food_1)
     11 print(food_2)
---> 12 print(food_1 < food_2)  #파이썬

TypeError: '<' not supported between instances of 'Food' and 'Food'
```

```
food_3 = Food('빙수', 8000)
food_2 = Food('햄버거', 5000)

# food_3가 food_2보다 작은지 확인하세요
print(food_3 < food_2)
```

```
False
```

실습

- Food class에서의 "+" (덧셈)은 다음의 예에서와 같이 객체들의 가격을 더하도록 수정하시오.

```
food_1 = Food('아이스크림', 3000)
food_2 = Food('햄버거', 5000)

# food_2가 food_1보다 큰지 확인
print(food_1 < food_2)  # 3000 < 5000

print(food_1 + food_2)
```

```
True
8000
```

13 요일 구하기 예제를 통한 객체지향 알아보기

- 요일구하기 예제에서 어떠한 객체가 필요한가?
- 요일 구하기 예제를 객체 지향으로 구현해 보자.