파이썬 프로그래밍 기초

모듈(Module)

파이썬 모듈

- 함수나 변수 등을 정의해 놓은 파이썬 프로그램 파일
- 모듈 내에서는 어떤 코드도 작성 가능하다 (변수, 함수, 클래스 등)
- 다른 모듈에 의해 호출되고 사용된다
- 모듈의 종류
  - 표준 모듈
  - 사용자 생성 모듈
  - 서드 파티 모듈

파이썬 모듈
: 간단한 모듈 만들기

```
# mymod.py

pi = 3.14159

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

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

def divide(a, b):
    return a / b
```

파이썬 모듈
: 모듈 불러오기 - import

```
import {모듈명}
```

```
# test-mymod.py

import mymod

print(mymod.add(10, 20))
print(mymod.subtract(10, 20))
print(mymod.multiply(10, 20))
print(mymod.divide(10, 20))
```

파이썬 모듈
: 네임스페이스

- 네임스페이스는 모듈 내부의 이름(변수, 함수, 클래스)를 구분하는 역할 수행
- 네임스페이스가 주어지지 않은 변수나 함수는 LEGB 규칙에 따라 찾게 된다

```
import math
import mymath

print(mymath.pi) # mymath 모듈 내의 pi 이용
print(math.pi) # math 모듈 내부의 pi 이용
```

파이썬 모듈
: from ~ import

```
from {모듈명} import {모듈객체} # 모듈명 없이 객체명만으로 접근
```

```
from math import pi, sin, con, tan
print(sin(pi/6), cos(pi/3), tan(pi/4))
```

- 현재 모듈에 특정 이름이 중복되는 경우 맨 마지막에 import된 객체가 적용

```
from math import pi, sin, con, tan
from mymod import pi
print(sin(pi/6), cos(pi/3), tan(pi/4))
```

- 모듈 내에 정의된 모든 이름을 현재 모듈로 가져온다(*)

```
from math import * # math 모듈 내 모든 이름 가져오기
print(sin(pi/6), cos(pi/3), tan(pi/4))
```

파이썬 모듈
: from ~ import

- 모듈 이름을 다른 이름으로 바꾸는 것도 가능

```
from math import *
import mymod as m # mymod의 이름을 m으로 변경
print(sin(pi/6), cos(pi/3), tan(pi/4)) # math 모듈 내의 객체들을 이용
print(m.pi) # mymod의 pi 객체를 이용
```

- 모듈 내에 정의된 객체의 이름을 변경하는 것도 가능

```
from math import sin as msin, cos as mcos, tan as mtan
import mymod as m # mymod의 이름을 m으로 변경
print(msin(m.pi/6), mcos(m.pi/3), mtan(m.pi/4))
```

모듈 지원 함수 목록 보기
: dir 함수

- dir 함수 인자에 객체를 넣어주면 해당 객체가 어떤 변수와 메서드를 갖고 있는지 반환해준다

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> a = [1, "Python", 3.14159]
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

파이썬 모듈
: __name__과 “__main__”

- 모든 모듈은 모듈의 이름을 저장하고 있는 내장 변수 __name__을 갖고 있다
- 최상위 모듈(인터프리터에서 실행되는 모듈)의 __name__ 은 “__main__” 이다
- 그 외 import 되는 모듈의 이름은 파일명이다

```
# mymod.py
print("mymod.py의 모듈이름:" + __name__)
```

```
# moduleimport.py
import mymod
print("moduleimport.py의 모듈이름:" + __name__)
```

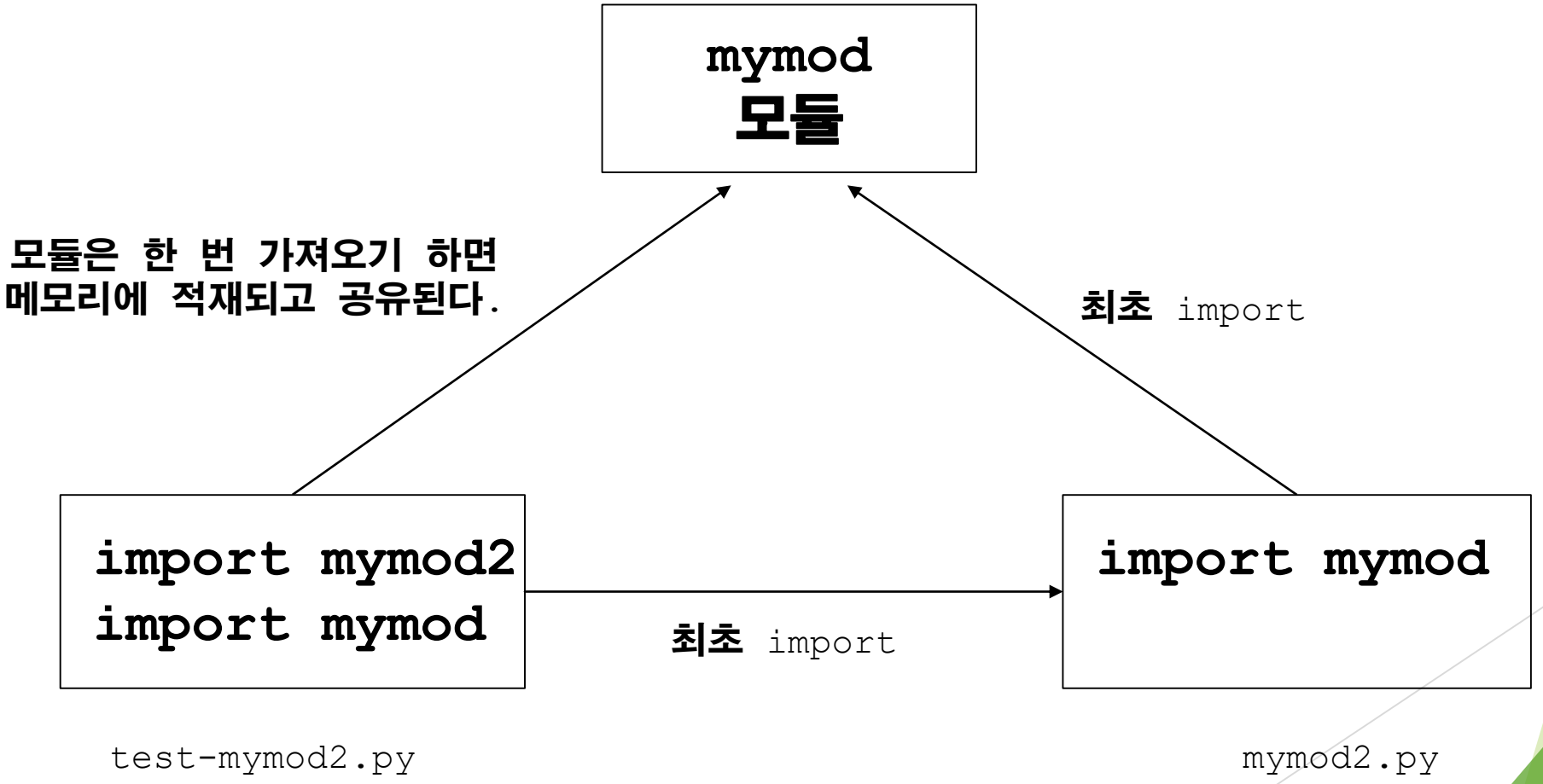
```
> python moduleimport.py
mymod.py의 모듈이름:mymod
moduleimport.py의 모듈이름:__main__
```

파이썬 모듈
: __name__과 “__main__”

- 내장 변수 __name__과 최상위 실행 모듈의 이름이 “__main__”인 점을 응용하면
  - import 되는 모듈로 사용하면서
  - 자신이 최상위 모듈로 실행될 때만 특정 기능을 수행하는 코드를 만들 수 있다

```
# mymod.py
def main():
    print("mymod.py를 최상위 모듈로 실행했습니다.")

if __name__ == "__main__":
    main()
else:
    print("mymod.py의 모듈이름:" + __name__)
```

파이썬 모듈
: 모듈의 공유


mymod
모듈
모듈은 한 번 가져오기 하면
메모리에 적재되고 공유된다.
최초 import
import mymod2
import mymod
최초 import
import mymod
test-mymod2.py
mymod2.py

파이썬 모듈
: import한 module 이름의 열거

- 한번이라도 import한 모듈은 dict 타입인 sys.modules 변수에 저장된다

```
import mod_a
import mod_b
import mymod
import mymod2
import sys

for key in sys.modules.keys():
    print(key)
```

파이썬 모듈
: 이름(변수, 함수, 클래스)이 속한 모듈 알아내기

- 파이썬 변수, 함수, 클래스는 각각 자신이 정의된 모듈의 이름이 저장된 __module__ 속성을 가지고 있다

```
>>> from math import sin
>>> from cmd import Cmd
>>> sin.__module__
'math'
>>> Cmd.__module__
'cmd'
```

여러가지 내장 모듈
: sys 모듈의 argv

```
import sys
```

- 파이썬 인터프리터와 관련된 정보와 기능 제공
- argv : 명령행에서 넘어온 인수(arguments)를 처리할 수 있다

```
# args.py
import sys
args = sys.argv[1:] # arg[0]는 스크립트명 자체

for x in args:
    print(x, end = " ")
else:
    print()
```

```
python args.py arg1 arg2 arg3
arg1 arg2 arg3
```

여러가지 내장 모듈
: math 모듈 - 파이와 자연상수

```
import math
```

- 파이와 자연상수

```
>>> math.pi # 파이값
3.141592653589793
>>> math.e # 자연상수
2.718281828459045
```

여러가지 내장 모듈
: math 모듈 - 절대값, 반올림, 버림계산

- abs, round는 내장 수치함수

```
>>> abs(10) # 절대값
10
>>> abs(-10) # 절대값
10
>>> round(1.2345, 2) # 소수점 셋째자리에서 반올림
1.23
>>> round(1.5213) # 소숫점 반올림
2
>>> math.trunc(1.7) # 소숫점 이하 버림
1
```

여러가지 내장 모듈
: math 모듈 - 팩토리얼, 제곱과 제곱근

```
>>> math.factorial(10) # 10! = 10 * 9 * 8 * ... 1
3628800

>>> math.pow(3, 3) # 3의 3승 = 3 * 3 * 3
27.0

>>> math.sqrt(4) # 제곱근
2.0
```

여러가지 내장 모듈
: math 모듈 - 로그 함수

- 첫 번째 매개변수의 로그를 반환, 두 번째 매개변수는 밑수
  - 두 번째 매개변수가 생략되면 밑수는 자연지수 e로 간주한다
  - 밑수가 10인 로그를 위한 log10 함수도 별도로 제공

```
>>> math.log(2)
0.6931471805599453
>>> math.log(4, 2)
2.0
>>> math.log10(1000)
3.0
```

여러가지 내장 모듈
: re 모듈 - 정규식(Regular Expression)

```
import re
```

- string 보다 더 전문적으로 문자열을 다룰 수 있는 모듈
- 문자열 내에서 패턴에 매칭되는 문자열을 추출

```
import re

list = """
Primary email : skyun.nam@gmail.com
Secondary email : litmuscube@gmail.com
"""
result_list = re.findall(r"(\w+[\w\.]*)@(\w+[\w\.]*)\.([A-Za-z]+)", list)
print (result_list)
for result in result_list:
    print (result)
```

여러가지 내장 모듈
: random 모듈 - 난수

```
import random
```

- 임의로 특정 값을 선택해 제공하는 기능 - 주사위의 예
- random 모듈은 단순히 난수를 발생하는 것 이외에 다양한 기능을 제공

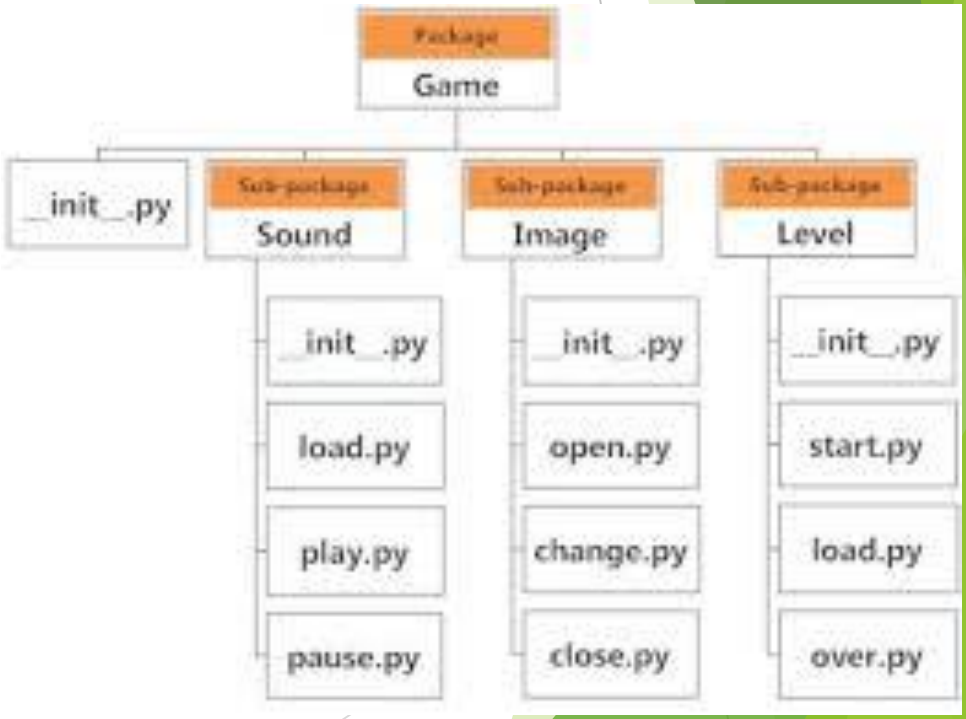
함수	설명
random()	0 ~ 1 사이의 난수를 발생
randint(s, e)	s ~ e 사이의 정수 난수를 발생
randrange([start,] stop[, step])	start, stop 간격 step 사이의 수 중에서 난수를 발생
shuffle(seqvar)	시퀀스 자료형을 섞음
choice(seqvar)	시퀀스 자료형에서 아무 거나 하나를 뽑아줌
sample(seqvar, size)	시퀀스 자료형에서 size만큼의 값을 임의로 뽑아옴

여러가지 내장 모듈
: random 모듈 - 난수

```
>>> import random
>>> random.random() # 0 ~ 1 사이의 난수를 발생
0.05414977057567982

>>> random.randint(1, 6) # 1 ~ 6 사이의 수 중에서 정수 난수를 발생
1

>>> random.randrange(1, 100, 3) # 1 ~ 100 사이 3간격의 수 중에서 난수 발생
40

>>> seqvar = ["짬뽕", "짜장면", "짬짜면"]
>>> random.shuffle(seqvar) # 시퀀스 자료형 섞기
>>> seqvar
['짬짜면', '짜장면', '짬뽕']

>>> random.choice(seqvar) # 시퀀스 자료형 중 임의로 한 개의 값 반환
'짜장면’
```

패키지

- 모듈을 모아놓은 단위
- 관련된 여러 개의 모듈을 계층적인 디렉터리로 분류해서 저장하고 관리한다
- 닷(.)을 이용하여 관리할 수 있다
- 오른쪽 예제에서 Game, Sound, Image, Level은 디렉토리이고 .py 파일이 파이썬 모듈이다
- __init__.py는 패키지를 인식시켜주는 역할을 수행
  -> 특정 디렉토리가 패키지로 인식되기 위해 필요한 파일



파이썬 프로그래밍 기초

클래스

파이썬 클래스

- 새로운 이름 공간을 지원하는 단위: 데이터의 설계도
- 데이터와 데이터를 변경하는 함수(메서드)를 같은 공간 내에 작성
- 클래스를 정의하는 것은 새로운 자료형을 정의하는 것이고, 인스턴스는 이 자료형의 객체를 생성하는 것이다
- 클래스와 인스턴스는 각자의 이름공간을 가지게 되며 유기적인 관계로 연결

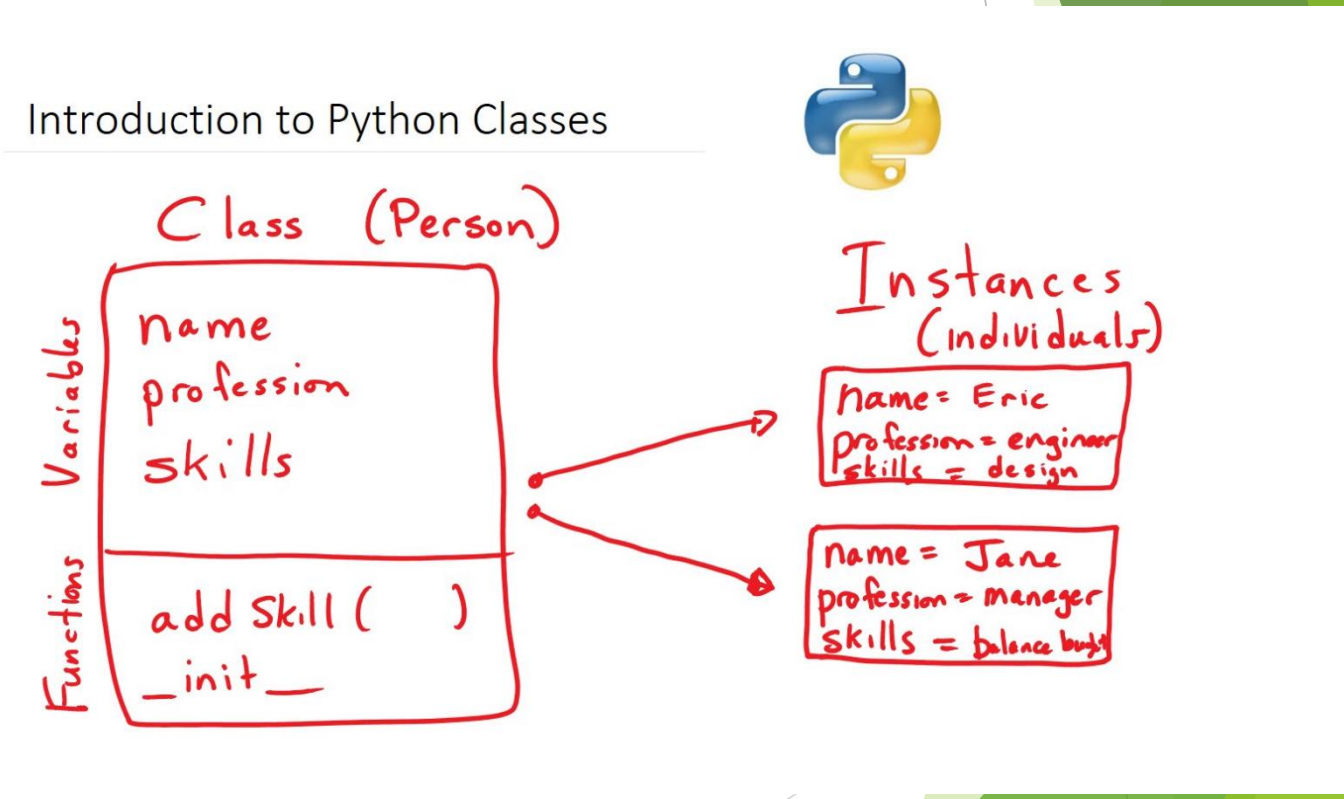
```
class MyString(str):
    pass

s = MyString()
print(type(s))
print(MyString.__bases__)

s2 = str()
print(type(s2))
print(str.__bases__)
```

파이썬 클래스
: 용어 정리

용어	설명
클래스(Class)	class 문으로 정의하며, 멤버와 메서드를 가지는 객체
클래스 객체	어떤 클래스를 구체적으로 가리킬 때 사용
인스턴스	클래스를 호출하여 만들어지는 객체
인스턴스 객체	인스턴스화 된 객체
멤버	클래스가 갖는 변수
메서드	클래스 내에 정의된 함수
속성	멤버 + 메서드
상위클래스	기반 클래스. 어떤 클래스의 상위에 있으며 여러 속성을 상속해준다
하위클래스	파생 클래스. 상위 클래스로부터 여러 속성을 상속 받는다

파이썬 클래스
: 클래스 이용의 장점

- 프로그램의 규모가 커졌을 때 의미 있는 집합체 단위로 프로그램을 정리할 수 있다
- 설계도(Class)가 있으므로 인스턴스를 양산할 수 있다



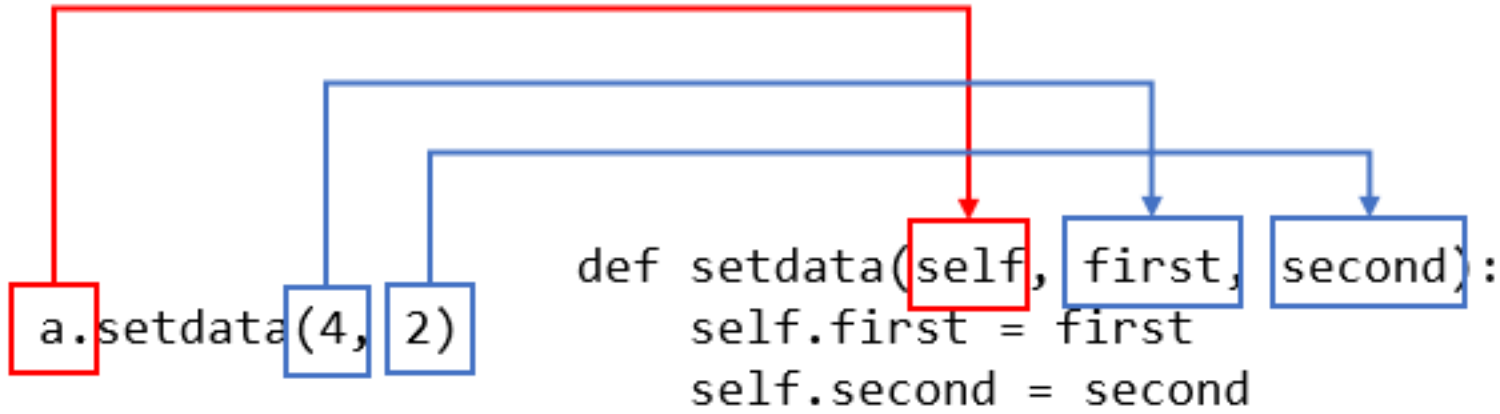
파이썬 클래스
: 클래스의 생성과 메서드의 정의

```
# point.py
class Point:

    def set_x(self, x):
        self.x = x

    def get_x(self):
        return self.x

    def set_y(self, y):
        self.y = y

    def get_y(self):
        return self.y
```

* 인스턴스 메서드의 첫 번째 인자는 항상 self

파이썬 클래스
: 인스턴스 객체 생성과 메서드 호출

- Bound Instance Method 호출




```
# paint.py
from point import Point

def main():
    bound_class_method()

def bound_class_method():
    p = Point() # Point 인스턴스 객체 생성
    p.set_x(10)
    p.set_y(10)
    print(p.get_x(), p.get_y(), sep = ',')

if __name__ == '__main__':
    main()
```

파이썬 클래스
: 인스턴스 객체 생성과 메서드 호출

- Unbound Instance Method 호출(참고)

```
# paint.py
from point import Point

def main():
    unbound_class_method()

def unbound_class_method():
    p = Point()
    Point.set_x(p, 10) # 객체를 첫 번째 인자에 할당하고 있음에 유의
    Point.set_y(p, 10) # 객체를 첫 번째 인자에 할당하고 있음에 유의
    print(Point.get_x(p), Point.get_y(p), sep = ',')

if __name__ == '__main__':
    main()
```

파이썬 클래스
: 정적 메서드(static method)와 클래스 메서드(class method)

- 인스턴스 객체의 멤버에 접근할 필요가 없는 메서드
- 첫 번째 인자로 인스턴스 객체 참조값을 받지 않는 클래스 내에 정의된 메서드
- Class 메서드의 첫 번째 인자는 클래스 객체 참조를 위한 객체 참조값
- @staticmethod, @classmethod 데코레이터로 손쉽게 구현 가능

```
# in point.py

# ...
    @staticmethod
    def static_method():
        return "static_method() 호출"

    @classmethod
    def class_method(cls): # => 클래스 참조를 위한 객체 참조값
        return "class_method() 호출"

# ...
```

파이썬 클래스
: 클래스 멤버와 인스턴스 멤버

종류	이름 공간	공유 범위
클래스 멤버	클래스 이름 공간 내	모든 인스턴스 객체들에 공유
인스턴스 멤버	인스턴스 이름 공간 내	개별 인스턴스 객체에서만 참조

```
# in point.py

class Point:
    count_of_instance = 0

    def set_x(self, x):
        self.x = x
```

클래스 멤버 정의

인스턴스 멤버 정의

파이썬 클래스
: 클래스 멤버와 인스턴스 멤버 접근

- 인스턴스 객체에서 참조하는 멤버의 객체를 찾는 순서는 아래와 같다
  - 인스턴스 멤버
  - 인스턴스 멤버가 없다면 클래스 멤버를 찾음

```
# in paint.py

def test_member():
    p = Point()
    Point.set_x(p, 10)
    Point.set_y(p, 10)
    print('x={0}, y={1}, count_of_instance={2}'.format(p.x, p.y, p.count_of_instance))
```

파이썬 클래스
: 생성자와 소멸자

- 생성자 : 클래스가 인스턴스화 될 때 실행되는 내용. 초기화.
  - __init__ 메서드 내에 작성
- 소멸자 : 클래스 인스턴스가 제거될 때 실행되는 내용.
  - __del__ 메서드 내에 작성

```
# in point.py

# ...
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y
        Point.count_of_instance += 1

    def __del__(self):
        Point.count_of_instance -= 1
# ...
```

파이썬 클래스
: __str__ 메서드

- 객체를 문자열로 반환하는 함수

```
# in point.py

# ...
    def __str__(self):
        return "Point({0}, {1})".format(self.x, self.y)
# ...
```

```
# in paint.py

def test_to_string():
    p = Point()
    print(p)

# => Point(0, 0)
```

파이썬 클래스
: __repr__ 메서드

- __str__과 비슷하지만 "문자열로 객체를 다시 생성할 수 있기 위해" 사용
- Eval을 수행하면 다시 그 해당 객체가 생성될 수 있어야 한다

```
# in point.py

# ...
    def __repr__(self):
        return "\"Point({0}, {1})\"".format(self.x, self.y)
# ...
```

```
# in paint.py

def test_to_string():
    p = Point()
    print(p)
    print(repr(p))

    p2 = eval(repr(p))
    print(p2)
```

파이썬 클래스
: __str__ vs __repr__

	__str__	__repr__
구분	비공식적 문자열 출력	공식적 문자열 출력
목적	사용자가 보기 쉽게	문자열로 객체를 다시 생성할 수 있도록
대상	사용자(End User)	개발자(Developer)

파이썬 클래스
: 연산자 재정의(Operator Overloading)

- 연산자에 대해 클래스에 새로운 동작을 정의하는 것
- 파이썬의 클래스는 새로운 데이터 형을 정의하는 것이므로 그에 상응하는 연산자의 재정의가 필요할 수 있다
- 연산자가 정의되어 있지 않으면 TypeError가 발생
- 파이썬에서는 사용하는 거의 모든 연산에 대해 새롭게 정의할 수 있다
  - 수치 연산자 오버로딩
  - 역이항 연산자 오버로딩
  - 확장 산술 연산자 오버로딩
  - 비교 연산자 오버로딩

파이썬 클래스
: 수치 연산자 오버로딩

연산자	연산자 메서드
+	__add__
-	__sub__
*	__mul__
/	__truediv__
//	__floormod__
%	__mod__
divmod()	__divmod__
pow(), **	__pow__
<<	__lshift__
>>	__rshift__
&	__and__
^	__xor__
|	__or__

- + 연산자 오버로딩 예제

```
class MyString:

    def __init__(self, s):
        self.s = s

    def __add__(self, other):
        return self.s + other

print(MyString("Life is too short, ") + "You need
Python!")
```

파이썬 클래스
: 역이행 연산자 오버로딩

연산자	연산자 메서드
+	__radd__
-	__rsub__
*	__rmul__
//	__rfloormod__
%	__rmod__
divmod()	__rdivmod__
pow(), **	__rpow__
<<	__rlshift__
>>	__rrshift__
&	__rand__
^	__rxor__
|	__ror__

- + 역이행 연산자 오버로딩 예제

```
class MyString:

    def __init__(self, s):
        self.s = s

    def __add__(self, other):
        return self.s + other

    def __radd__(self, other):
        return other + self.s

print(MyString("Life is too short, ")
    + MyString("You need Python!"))
```

파이썬 클래스
: 확장 산술 연산자 오버로딩

연산자	연산자 메서드
+=	__iadd__
-=	__isub__
*=	__imul__
//=	__ifloormod__
/=	__idiv__
%=	__imod__
**=	__ipow__
<<=	__ilshift__
>>=	__irshift__
&=	__iand__
^=	__ixor__
|=	__ior__

- +=, -= 연산자 오버로딩 예제

```
# in point.py
    def __iadd__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __isub__(self, other):
        return Point(self.x - other.x, self.y - other.y)
```

```
>>> from point import Point
>>> p = Point(10, 10)
>>> print(p += Point(20, 15)
>>> p += Point(20, 15) # __iadd__ 메서드로 구현된 += 연산자
>>> print(p)Point(30, 25)
```

파이썬 클래스
: 비교 연산자 오버로딩

- ==, <, > 연산자 오버로딩 예제

연산자	연산자 메서드
<	__lt__
<=	__le__
>	__gt__
>=	__ge__
==	__eq__
!=	__ne__

```
# in point.py
class Rect:
    def __eq__(self, other):
        return self.area() == other.area()

    def __lt__(self, other):
        return self.area() < other.area()

    def __gt__(self, other):
        return self.area() > other.area()
```