0127

** Function

〈 가변 매개변수 〉

(Arbitrary Argument List , Variable args)

: 매개변수의 개수가 정해지지 않아서
함수를 호출할 때 원하는 만큼의 매개변수 를 대입하는 형태

〈 형태 〉

함수를 정의할 때
* 이름 또는
**이름 을 주는 형태로 생성

1. * 이름 : tuple 형태로 만들어진다
2. **이름 : dict 형태로 만들어진다

〈 특징 〉

가변 parameter 앞에 있는
parameter의 값을 대입할 때는
이름과 함께 대입할 수가 없음

가변 parameter 뒤에 있는
parameter의 값을 대입할 때는
반드시 이름과 함께 대입해야 한다

** 을 이용해서 만들어진 parameter 는
parameter 중에서 가장 뒤에 위치
이름과 함께 대입 해야함
(사용시에 document 를 찾아보는 것이 좋다) ——〉 help 만으로는 부족

이 형태를 사용하는 대표적인 함수
시각화 함수

그래프들의 옵션 설정시 이 형태의 parameter 를 많이 사용

```python
# 가변 parameter
# 함수의 parameter 에
# * 이 붙어 있으면
# 가변 parameter 이다
# 즉, parameter 의 개수가 정해 지지 않았음 을 의미

def varargs1(*arglist):
    for arg in arglist:
        print(arg)


# 함수에 호출
varargs1("hi", "today is january 27th", "Thursday")
print()
varargs1("hi today is january 27th", "Thursday")
print()
varargs1("hi today is january 27th Thursday")


def varargs21(arg, *arglist1):
    print(f"일반 parameter :{arg}")
    print()
    for arg in arglist1:
        print(arg)


varargs21("hi", "my", "name", "is", "mimi")
# 첫번째 parameter 는 arg 에 대입 되고
# 나머지는 arglist1 에 대입
```

```
hi
today is january 27th
Thursday

hi today is january 27th
Thursday

hi today is january 27th
Thursday
일반 parameter :hi

my
name
is
mimi
```

error

```python
## 가변 parameter 앞의 데이터에는 이름을 사용할 수 없다
varargs21(arg="hi", "my", "name", "is", "mimi")
```

```
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 36
    varargs21(arg="hi", "my", "name", "is", "mimi")
                                            ^
SyntaxError: positional argument follows keyword argument
```

File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 36
varargs21(arg="hi", "my", "name", "is", "mimi")
^
SyntaxError: positional argument follows keyword argument

```python
def varargs3(*arglist3, arg):
    for arg in arglist3:
        print(arg)
    print(f"일반 parameter:{arg}")


# 가변 parameter 뒤의 parameter 는 반드시 이름과 함께 대입
varargs3("one", "two", "three", arg="number")

# 가변 parameter 뒤의 parameter 는 이름과 함께 입력되지 않으면 Error
varargs3("1", "2", "3", "4")
```

```
일반 parameter:n?umber
one
two
three

Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 49, in <module>
    varargs3("1", "2", "3", "4")
TypeError: varargs3() missing 1 required keyword-only argument: 'arg'
```

```python
# ** 이 붙으면 dict
# ** parameter 는 가장 마지막에 위치해야한다 .
def userURLBuilder(server, port, **param):
    uri=f"https://{server}:{port}/"
    #dict 는 순회를 하면 key 값이 return
    queryString = ""
    for attr in param:
        queryString += f"{attr}={param[attr]}&"
    uri += queryString
    print(uri)

# ** 에 값을 대입할 때는 이름과 값을 같이 준다
# 이름은 개발자가 정한 이름을 사용해야한다
userURLBuilder("localhost", "8080", name="sinsiuk", nickname="singsi")
```

```
https://localhost:8080/name=sinsiuk&nickname=singsi&
```

```python
# ** 다음에 새로운 parameter를 만들면 error
def userURLBuilder0(**param, server1, port1):
    print("success")
```

```
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 36
    varargs21(arg="hi", "my", "name", "is", "mimi")
                                                   ^
SyntaxError: positional argument follows keyword argument
```

〈 재귀 함수 〉

- recursion , recursive call
: 함수가 자기 자신을 다시 호출하는 경우

〈 장점 〉
코드가 간결해지고 이해하기 쉽도록 해준다

〈 단점 〉
메모리를 많이 사용하고 수행속도가 느리다

——〉 합계 , factorier , 피보나치 수열, 하노이의 탑 등이 대표적

```
def 함수이름(parameter):
        if 중단 조건:
            return 값
        return 함수이름(parameter)
```

Ex)
피보나치 수열
—〉 1, 1, 2, 3, 5, 8, ...
첫째, 둘째 항은 무조건 1.
세번째 항 부터는 이전 두 개 항의 합

재귀함수
피보나치 수열의 값을 리턴하는 함수

```python
def fibonacci(x):
    if x == 1 or x == 2:
        return 1
    else:
        return fibonacci(x - 1) + fibonacci(x - 2)


print(fibonacci(10))
```

55

〈 Python에서 함수는 일급 객체 〉

: 함수도 하나의 자료형이다

- 함수를 변수에 대입하는 것이 가능하다.

```
# 함수는 일급 객체
## 함수를 변수에 대입할 수 있다 .

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


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


# 함수를 변수로 지정
x = plus
y = minus

# 변수를 통해서 함수를 호출
print(x(1, 1))
print(y(200, 100))
```

```
2
100
```

- 함수를 매개변수로 사용할 수 있음
: 자바는 Event 처리할 때 객체를 대입
python은 함수를 대입

```
# 이 함수는 함수를 parameter 로 받아서 함수가 수행되고
# return
def click(func, n1, n2):
    return func(n1, n2)


print(click(x, 200, 50))
```

```
250
```

- 함수를 Return 할 수도 있다.

closure
: 변수를 스코드 안에 가둬두고 외부에서 사용하는 개념
함수의 지역변수를 함수 외부에서 사용하는 것을 말함

함수 Vs Method
- 함수는 전역 영역에 배치가 된다.
함수를 return 하게 되면 return 이 함수
return 내용이 전역 영역에 남아있어
이전의 내용을 사용할 수 있게 된다
이러한 전역 영역을 보통 Static 영역이라 한다

```
# Closure – 함수 내의 변수를 외부에서 변경할 수 있도록 하기 위함
def outer():
    x=0
    # 함수 안의 내용이 있다고 가정
    print(x)

    #리턴할  함수를  생성 - Local 이나 Grobal 이라는 예약어 사용 시
    내용  변경이 가능
    def inner():
        print(x)

    return inner

f = outer() # outer 가 return 한 inner 함수를 f 에 대입
print()
f()
```

```
0

0
```

〈 Pass 〉

함수 나 클래스의 내용을 작성하지 않을 때

이름만 만들어 두기 위해 사용 (Error 를 제거하기 위해)

〈 함수의 도움말 만들기 〉

: 함수 내부

""" """ 나 ''' ''' 사이에 내용을 기재

함수 외부

doc 속성에 문자열을 대입

```
< 도움말 만들기 >
def outer():
    '''
    이 함수는 클로저를 알아보기 위해서 작성하였습니다
    '''
    # 함수 안의 내용이 잇다고 가정
    print(x)

    def inner():
        print(x)

    return inner

def testhl():
    print("hello")

f = outer()  # outer 가 return 한 inner 함수를 f 에 대입
print()
f()


testhl.__doc__ = '''
이 내용은 help를 했을 때 보여지는 내용입니다.
'''

help(outer)
help(testhl)
```

```
outer()
    이 함수는 클로저를 알아보기 위해서 작성하였습니다

Help on function testhl in module __main__:

testhl()
    이 내용은 help를 했을 때 보여지는 내용입니다.
```

〈 lambda 〉

: 이름 없는 한 줄 짜리 함수

이름이 없다

——〉 static 영역에 저장이 안됨(바로 사용하고 없어 짐)

주로 함수를 parameter 로 대입해야 할 때 주로 사용

〈 작성 방법 〉

lambda 인수 나열 : 리턴할 내용

〈 특징 〉

매개변수 초기화 가능

이미 만들어진 변수 사용 가능

새로운 변수 생성은 안됨 (한줄 만 생성 되기 때문)

```
# 람다 사용
# 2개의 정수를 parameter 로 받아서 더한 결과를 리턴하는 함수
def plus(a, b):
    return a + b


# 람다로 만들기
f = lambda a, b: a + b

# plus 함수와 위의 람다는 같다

print(f(100, 200))
```

300

```
# -10 부터 9 까지 1씩 증가하면서 func 에 대입해서나온 결과를
# list 를 생성하여 return 하는 함수

def g(func):
    return [func(x) for x in range(-10, 10, 1)]

# 이때 parameter 로 lambda를 많이 사용

print(g(lambda x : x*x))
```

[100, 81, 64, 49, 36, 25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

〈 함수형 프로그래밍에서 사용되는 함수 〉

〈 map 〉
: 데이터의 모임 과 parameter 가 1개 이고
return 하는 함수를 받아서 데이터 모임의 모든 데이터를
순회하면서 parameter로 대입된 함수를 결과로
데이터의 모임을 만들어주는 함수

——〉 데이터의 모임을 받아서 필요한 연산을 수행
수행한 결과로 데이터의 모임을 만들어 주는 함수

——〉 반복문을 사용하는 것 보다 빠른속도로 연산을 수행

```python
# map 함수 -- 데이터를 변환해 주는 함수

import datetime

# 0 ~ 9999 까지 list 를 생성
li = [i for i in range(10000)]

# 현재 시간 구하기

# 1. 일반 반복문을 사용
# 시작 시간
s1 = datetime.datetime.now()
for i in li:
    print(i * i, end=" ")

# 종료 시간
s2 = datetime.datetime.now()
print()
print("일반 반복문 처리 시작 시간 : ", s1)
print("일반 반복문 처리 종료 시간 : ", s2)
print("일반 반복문 처리 소요 시간 : ", s2 - s1)


def func00(x):
    return x * x


# 2. 함수를 이용한 처리
# 시작 시간
s3 = datetime.datetime.now()

for case in li:
    print(func00(case), end=" ")

# 종료 시간
s4 = datetime.datetime.now()
print()
print("함수를 이용한 처리 시작 시간 : ", s3)
print("함수를 이용한 처리 종료 시간 : ", s4)
print("함수를 이용한 처리 소요 시간 : ", s4 - s3)

# 3. map과 lambda를 이용하는 경우

# 시작 시간
s5 = datetime.datetime.now()

print(map(lambda x1: x1 * x1, li))

# 종료 시간
s6 = datetime.datetime.now()
print()
print("map 과 lombda 를 이용한 처리 시작 시간 : ", s5)
print("map 과 lombda 를 처리 종료 시간 : ", s6)
print("map 과 lombda 를 처리 소요 시간 : ", s6 - s5)


print("일반 반복문 처리 소요 시간 : ", s2 - s1)
print("함수를 이용한 처리 소요 시간 : ", s4 - s3)
print("map 과 lombda 를 처리 소요 시간 : ", s6 - s5)
```

```
일반 반복문 처리 소요 시간 :  0:00:00.019937
함수를 이용한 처리 소요 시간 :  0:00:00.021443
map 과 lombda 를 처리 소요 시간 :  0:00:00.000008
```

〈 filter 〉

: 데이터의 모임 과 parameter 가 1개이고
bool 을 리턴하는 함수를 이용해서
데이터의 모임에서 true 를 리턴하는 데이터만 골라내는 함수

```
li1 = [1, 2, 3, 4, 5, 6, 7]

# 짝수 데이터만 골라내는 작업 실행하기
data = []
for temp in li1:
    if temp % 2 == 0:
        data.append(temp)
print(data)

# filter 를 사용해서 출력하기
print()
print([filter(lambda x2: x2 % 2 == 0, li1)])
print()
print(list(filter(lambda x2: x2 % 2 == 0, li1)))
```

```
[2, 4, 6]

[<filter object at 0x7f8fc81dd6d0>]

[2, 4, 6]
```

〈 reduce 〉

: 데이터의 모임과 parameter 가 2개인 함수를 가지고
집계를 해서 하나의 결과를 return

앞의 parameter 는 연산의 결과
뒤의 parameter 는 데이터

python 에서 기본 함수였지만
최근 function tools 모듈로 이전

```
from functools import reduce

result = reduce(lambda x3, y3: x3 + y3, [100, 200, 300, 700])
print(result)
```

```
1300
```

〈 Map-Reduce 〉

: 분산 환경에서 테스크를 클러스트들에
나누어서 처리한 후 결과를 집계하는 방식의 FrameWork

〈 Scope 〉

변수의 유효 범위
- python 은 함수 안에 함수를 생성할 수 있음
- 함수 안에서 만들어진 변수는 함수안에서만 사용 가능

함수 내에서 외부에 있는 변수를 사용하고자 할때
변수 이름 앞에 global을 붙이면 된다 .

```
print("scope 변수의 유효 범위")
# scope 변수의 유효 범위
def outer():
    x = 10
    print(x)
    def inner():
        x = 20
        print(x)
    inner()
    print(x)

outer()
```

```
10
20
10
```

```
print("scope 변수의 유효 범위")
# scope 변수의 유효 범위
def outer():
    x = 10
    print(x)
    def inner():
        # x = 20
        print(x)
    inner()
    print(x)

outer()
```

```
10
10
10
```

```
print("scope 변수의 유효 범위")
# scope 변수의 유효 범위
def outer():
    x = 10
    print(x)
    def inner():
        # 이제부터 x는 포함한 함수에 있는 x
        # 선언 해줌으로서 지역화시킴
        nonlocal x
        x = 20
        # inner 의 x 를 출력
        # inner 에 x 가 없다면 바깥에 있는 x를 출력
        print(x)
    inner()
    # 여기서는 outer 의 x인 10이 출력
    print(x)

outer()
```

```
scope 변수의 유효 범위
10
20
20
```

〈 Python 의 함수는 일급 객체 〉

변수나 자료구조에 함수를 대입할 수 있다.

- parameter 로 함수를 대입할 수 있다.

- return 값으로 함수를 사용할 수 있다.

- 함수를 실행 중에 만들 수 있다 .
  함수안에 함수를 만들거나 이름없는 함수(lambda)를 생성할 수 있다 .

** OOP(Object Oriented Programming) - 객체 지향 프로그래밍

- Compile Time : 실행 될 수 있는 코드를 만드는 시점
- Run Time : 실행 될 때

〈 Static Binding & Dynamic Binding 〉

Static Binding(정적 바인딩)
: Compile Time 에 결정하는 것

Dynamic Binding(동적 바인딩)
: Runtime 에 결정하는 것
(python , javascript —〉 그래서 변수 생성시 자료형을 기재하지 않음

〈 Encapsulation - 캐슐화 - 〉

—〉 불필요한 정보는 숨기고 중요한 정보만 외부로 노출하는 것
(Information Hiding - 정보 은닉 -)

연관된 작업(Method) 과
데이터(property, Attribute, Field)를
묶어서 표현

클래스를 만드는 것과 Instance를 만드는 작업

〈 Inheritance - 상속 - 〉

상위 클래스의 모든 것을
하위 클래스가 물려 받는 것

- 직접 구현하는 경우
여러 클래스들에 유사한 코드가 존재하는 경우
그 코드를 상위 클래스에 만들고
하위 클래스에서 상속 받아서 사용

- Frame work 는
자주 사용하는 Class 를 만들어두고
상속 받아서 사용하는 형태 (Sub Classing)

상위 : super , based
하위 : sub , Derived

〈 Polymorphism - 다형성 - 〉

: 동일한 message 에 대해서 다르게 반응하는 성질

동일한 코드가
    어떤 Instance 가 대입되어 있는지에 따라
        다른 method 를 호출하는 것을 말함

상속 과 method Overriding 을 이용해서 구현

〈 Class 〉

Property, Attribute, field (속성)
: Data를 저장하기 위한 목적을 가진 요소

method
: 한 번에 수행되어야 하는 동작을 가진 요소

class 클래스 이름 :
    속성 나열
    method 나열

〈 Instance - 객체 -〉

클래스를 기반을 해서 생성된 객체
    python 은 Dynamic Binding 이어서
        Instance를 생성 후, Instance 에 요소를 추가하는 것이 가능

(Java 에서는 Instance 생성하면 Class 와 모양이 같음)

Instance 생성
    클래스이름(parameter)

〈 속성이나 method 접근 〉

클래스 안에서는 이름만으로 접근 가능
클래스 외부에서는
        Class.
        instance.
            을 붙여서 접근

〈 클래스 안에 method 선언 〉

— 클래스 외부에서 Instance를 통해서 접근하는 method 생성 시
        무조건 parameter 가 1개 이상 있어야하고
            첫번째 parameter 는 무조건 Instance 자신임

관례상 이 첫번째 parameter 이름이 self

Class 안에 만든
Instance method 를 호출하는 방법

〈 방법 〉

Instance 이름 . method이름(두번째 parameter 부터 나열 …) : 바운드 호출

Class 이름 . method이름(Instance. 두번째 parameter 부터 나열 …) : 언바운드 호출

```python
# OOP // 클래스와 Instance 생성 및 Instance method 호출

# 클래스 생성
class Student:
    # Instance method 생성을 python 에서 할 경우
    # parameter 가 최소 1개는 있어야 한다
    # 첫번째 parameter는 Instance 자신이어야 한다
    def method1(self):
        print("method")

    def method2(self, name):
        print(f"name : {name}")


# Instance 생성
student = Student()

# 1. instance 를 이용한 method 호출 : 바운드 호출
student.method1()

# 2. 클래스를 이용한 instance method 호출 : 언바운드 호출
Student.method1(student)


# parameter 가 두개지만
# 첫번째 parameter 는 자기 자신을 가리키기 때문에
# 첫번째 paraemter 를 제외한 parameter 값을 넣어 줘야함
student.method2("가오나시")
```

```
method
method
name : 가오나시
```

〈 속성 생성 〉

클래스 안에서 method 외부에 변수를 선언하면
이 변수는 클래스의 속성이 된다 .

: 이 속성은 클래스를 통해 읽을 수 있고
Instance 를 통해서 읽을 수도 있다
이렇게 만든 속성에 클래스를 통해서 값을 대입하면
클래스의 속성에 값을 대입한 것

Instance 를 통해서 값을 대입하는 경우
Instance 가 소유하는 속성을 새로 만들어서 값을 대입

Instance를 통해서는 클래스 속성의 값을
변경할 수 없다 .

```python
class Student:
    # 클래스의 속성이 된다
    # 클래스를 통해서 읽을수도 있고
    # Instance를 통해서도 읽을 수 있다
    name = "default"


# 클래스를 이용해서 클래스 속성에 접근
print(Student.name)

# Instance 생성
# instance 생성 , () 를 열고 닫기 !
instance = Student()

# Instance 를 이용해서 클래스 속성에 접근
print(instance.name)

# Caution
# 클래스를 이용해서 클래스 속성을 수정
Student.name = "Class Attribute !"
print("클래스를 이용해서 클래스 속성을 수정")
print(Student.name)
print(instance.name)

# Instance 를 사용 해서 수정 했을 때
# Instance 에 name 속성을 만들어서 대입 -- > class 속성은 아무런 변화가 없다.
instance.name = "Instance Attritbute"
print("Instance 를 사용 해서 수정")
print(Student.name)
print(instance.name)
```

```
default
default
클래스를 이용해서 클래스 속성을 수정
Class Attribute !
Class Attribute !
Instance 를 사용 해서 수정
Class Attribute !
Instance Attritbute
```

—— 〉 모든 인스턴스가 공통으로 소유하는 속성을 갖고자 할때는
Instance method 안에서
Instance.속성이름 의 형태로 만들어야 한다 .

〈 접근자 method 〉
- Getter, Setter , Accessor

Instance의 속성에 접근하는 method

(Getter)
: 속성의 값을 return 해주는 method

- 형태 -
  get속성이름

: 속성이름의 첫글자는 대문자로 하거나, _를 추가하고 속성 이름을 기재

: 속성의 **자료형이 bool**인 경우 , get 대신에 is를 사용

: method 의 parameter 는 Instance 자신외에는 없고
내용은 속성의 값을 Return

(Setter)
: 속성의 값을 변경하는 methdod

- 형태 -
  set속성이름

: 속성의 **자료형이 bool** 이어도, Set 으로 시작

: method 의 parameter 은
1. Instance 자신
2. 변경할 데이터

내용
: parameter 값을 속성에 대입하는 것

속성이 집단 자료형인 경우
Index 나 Key 를 parameter 로 받아서
index , key 의 데이터를 Return 혹은 설정하는 method 를 만들기도 한다.

```python
# name 과 score 를 속성으로 갖는 instance 들의 class
class Student:
    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score


# 인스턴스 생성
nameScoreInstance = Student()
nameScoreInstance.setName("Ice Americano")
nameScoreInstance.setScore(4600)

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())
```

Ice Americano : 4600

〈 특수한 용도로 사용되는 속성 〉

__ 내용 __

ex)
__doc__ : 함수를 설명하는 문자열

__name__ : 함수의 이름

: 위와 같은 속성들은 함부로 사용하면 좋지 않다

- python에서는
  속성이나 method 이름으로 '_'로 시작하는 것을 권장하지 않는다.

〈 Constructor - 생성자 - 〉

: 초기화 method

Instance 가 생성될 때 호출되는 method

- 이름 -

__init__

생성하지 않으면

parameter 가 instance 자기 자신만 있는

초기화 method 가 만들어 진다

값을 대입받아서

속성들을 초기화 하거나

Instance 를 생성할 때 속성을 만들고자 하는 경우 생성

여러개 생성이 가능하다

〈 소멸자 〉

: Instance 가 소멸될 때 호출되는 method

- 이름 -

__del__

Instance 자신을 parameter 로 하는 method 하나만 생성 가능

```python
# name 과 score 를 속성으로 갖는 instance 들의 class
class Student:
    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score


# 인스턴스 생성
nameScoreInstance = Student()

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())

# setter 를 지우고 실행
# error  -- 지정된 Setter 값이 없어서
```

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 369, in <module>
    print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 354, in getName
    return self.name
AttributeError: 'Student' object has no attribute 'name'
'
```

```python
class Student:
    # 초기화 method self 이외의 parameter 가 없는 초기화 method
    # 속성의 값을 기본값으로 만들기 위한 경우
    def __init__(self):
        self.name = "noname"
        self.score = 0
    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score


# 인스턴스 생성
nameScoreInstance = Student()

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())
```

noname : 0

```python
class Student:
    # 초기화 method self 이외의 parameter 가 없는 초기화 method
    # 속성의 값을 기본값으로 만들기 위한 경우

    # Spring 에서 NoArgsConstructor
    def __init__(self):
        self.name = "noname"
        self.score = 0

    # Spring 에서 AllArgsConstructor
    def __init__(self, name, score):
        self.name = name;
        self.score = score;

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score


# 인스턴스 생성
# nameScoreInstance = Student()
nameScoreInstance = Student(name="ice", score=20)

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())
```

ice : 20

——〉 이 경우 parameter 가 없을 경우는 사용할 수 없음

```python
class Student:
    # 초기화 method self 이외의 parameter 가 없는 초기화 method
    # 속성의 값을 기본값으로 만들기 위한 경우

    def __init__(self, name="이름", score = 0):
        self.name = name
        self.score = score

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score


# 인스턴스 생성
nameScoreInstance = Student()

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())

nameScoreInstance = Student(name="ice", score=20)

print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())
```

이름 : 0
ice : 20

—— parameter 유무에 상관없이 사용할 수 있음

```
def __init__(self, name="이름", score=0):
        self.name = name
        self.score = score

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

    def getScore(self):
        return self.score

    def setScore(self, score):
        self.score = score

    def __del__(self):
        print("Instance will delete soon")


# 인스턴스 생성
nameScoreInstance = Student()
print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())

# ----> 새로운 객체를 만들게 되고 이전 객체는 가리키는 변수가 없으므로 소멸

nameScoreInstance = Student(name="ice", score=20)
print(nameScoreInstance.getName(), ":", nameScoreInstance.getScore())

# setter 를 지우고 실행
# error  -- 지정된 값이 없어서

# 어떤 Instance 를 소멸시키고자 하는 경우
# 1. Instance를 가리키는 변수에 다른 Instance 를 대입하거나
# 2. None 을 대입해주면 된다
```

```
이름 : 0
Instance will delete soon
ice : 20
Instance will delete soon
```

〈 인스턴스 소멸 〉

: Python 은 개발자가 직접 메모리를 해제 할 필요가 없다

Garbage Collection 이 대신 해줌

——〉 사용 방법
= retain count (참조 횟수)

Instance를 생성하면
retain count 는 1 이 된다 .

변수가 소멸되거나
변수에 None 을 대입하면

——〉 retain count 는 1 감소

retain count 가 0 이 되면 Instance는 소멸

Instance를 다른 변수가 또 가리키면
retain count 는 1 증가 한다.

retain count 를 확인하고자 할때
sys 모듈의
getrefcount 라는 함수에
Instance를 대입하면
retain count 를 리턴 해줌 (원래값 보다 1 증가한 값)

because : getrefcount 함수 자체가 Instance를 참조하기 때문

————〉 코드가 길어야 어느정도 확인이 가능함

〈 클래스가 Instance 생성 없이 호출 가능한 method 〉

〈 static method 〉

클래스 안에 method 생성 시
위에 @staticmethod 라고 기재
———〉 Instance를 parameter로 갖지 않는 method 를 생성할 수 있다.

(해당 기능 (@staticmethod) 를 python 에서는 decorator 라고 한다)

〈 class method 〉

클래스 안에 method 생성 시
위에 @classmethod 라고 기재
———〉 클래스 자신을 parameter로 갖는 method 생성 가능

```
class Coffee:
    def staticmethod():
        print("static method")


Coffee.staticmethod()
instanceCoffee = Student()
instanceCoffee.staticmethod()

#  - --- > parameter  가 하나 이상 있어야하기 때문에 작동이 안됨
```

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 403, in <module>
    instanceCoffee.staticmethod()
AttributeError: 'Student' object has no attribute 'staticmethod'
```

```
class Coffee:
    # Instance 생성 없이 클래스가 호출가능한 method 가 된다.
    @staticmethod
    def staticmethod():
        print("static method")


Coffee.staticmethod()
```

```
static method
```

```python
class Coffee:
    # Instance 생성 없이 클래스가 호출가능한 method 가 된다.
    @staticmethod
    def staticmethod():
        print("static method")

    # Instance 생성 없이 클래스가 호출가능한 method 가 된다
    # 이 코드가 있을 때, parameter를 하나 만들어야하는데 , 이 parameter 는
    # class 에 대한 정보를 가지고 있다.

    @classmethod
    def classmethod(cls):
        print("class method")


Coffee.staticmethod()
Coffee.classmethod()
```

static method
class method

〈 __slots__〉

: python 의 Instance는
정적이지 않고 동적이라서
생성한 후에도 속성을 추가할 수 있다.

— 이렇게 되면 Instance의 속성에 제한이 없어서
동일한 class 로 부터 만들어진 Instance 라도
가지고 있는 속성이 다르게 되서
하나의 Class 로 부터 만들어진 의미가 퇴색된다.

속성에 제한을 두고자 할 때
Class 내부에 __slots__ 속성에
속성 이름을 list 로 설정하면 된다
(값을 list 로 설정)

```
# DTO Class 먼저 만들어보기
class VO:
    def __init__(self, num=0, name="이름이 없습니다", score=0):
        self.num = num
        self.name = name
        self.score = score

    # getter setter 만들기 ---> 6 개를 만들어야 함 생략


vo1 = VO()
print(vo1.num, ":", vo1.name, ":", vo1.score, "점")
```

0 : 이름이 없습니다 : 0 점

```python
class VO:
    def __init__(self, num=0, name="이름이 없습니다", score=0):
        self.num = num
        self.name = name
        self.score = score

    # getter setter 만들기 ---> 6 개를 만들어야 함 생략



vo1 = VO()
print(vo1.num, ":", vo1.name, ":", vo1.score, "점")

# 기존 클래스에서 정의한 속성 이외의 속성 추가
vo1.grade = 3
print(f"번호 : {vo1.num}", f"이름 : {vo1.name}", f"점수 : {vo1.score}", f"학년 : {vo1.grade}")
```

번호 : 0 이름 : 이름이 없습니다 점수 : 0 학년 : 3

속성에 제한을 줘야한다.

```python
class VO:
    def __init__(self, num=0, name="이름이 없습니다", score=0):
        self.num = num
        self.name = name
        self.score = score

    # getter setter 만들기 ---> 6 개를 만들어야 함 생략

     # 이러면 미리 설정한 의미가 없고 , 값이 달라짐 -- > 속성에 제한을 줘야한다.
    __slots__ = ["num", "name", "score"]


vo1 = VO()
print(vo1.num, ":", vo1.name, ":", vo1.score, "점")

# 기존 클래스에서 정의한 속성 이외의 속성 추가
vo1.grade = 3
print(f"번호 : {vo1.num}", f"이름 : {vo1.name}", f"점수 : {vo1.score}", f"학년 : {vo1.grade}")
```

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 432, in <module>
    vo1.grade = 3
AttributeError: 'VO' object has no attribute 'grade'
```

———〉 모두 동일한 속성을 갖게 하기 위해서 사용 (__slots__)

〈 Private 〉

객체 지향 프로그래밍에서는

외부에서 접근할 필요가 없는 데이터는
private 을 사용해서 생성하는 것을 권장
(추상화)

객체 지향 프로그래밍에서는

외부에서 속성에 직접 접근하는 것을 권장하지 않음
——〉 Transaction 처리를 하지 못함
(method를 통해서 접근을 통해 보안유지 등 많은 이점들이 있다)

Python 에는 접근 지정자가 없다 .
대신에 속성을 만들 때 앞에
__ 추가 시 외부에서 접근할 수 없는 속성이 됨

- property -

속성에 method 를 할당해서
속성을 호출하면
get method 가 호출 되고
속성에 값을 대입하면
set method 가 호출

삭제하면 del method 호출 되도록 해주는 문법

속성명 = property(fget = get메서드 이름 , fset = set메서드이름 , fdel = del메서드 이름)

```python
class Rich:
    def __init__(self, name="singsiuk", budget="100 billion $"):
        self.name = name
        self.budget = budget


rich = Rich()

print(rich.name)
print(rich.budget)
```

```
singsiuk
100 billion $
```

————————〉 외부에서 접근이 불가능하도록 만들기

```python
class Rich:
    def __init__(self, name="singsiuk", budget="100 billion $"):
        self.name = name
        self.budget = budget


rich = Rich()

print(rich.__name)
print(rich.__budget)
```

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/0127_3rd.py", line 445, in <module>
    print(rich.__name)
AttributeError: 'Rich' object has no attribute '__name'
```

```python
class Rich:
    def __init__(self, name="singsiuk", budget="100 billion $"):
        self.name = name
        self.budget = budget

    # Getter Setter 생성
    def getName(self):
        print("name 의 getter 호출 ")
        return self.__name

    def getBudget(self):
        print("budget 의 getter 호출 ")
        return self.__budget

    def setName(self, name):
        self.__name = name

    def setBudget(self, budget):
        self.__budget = budget

rich = Rich()
rich.setName("singsi")
rich.setBudget("1000억")
print(rich.getBudget(), rich.getName())
```

budget 의 getter 호출
name 의 getter 호출
1000억 singsi

———————— >사용하기 힘들다

```python
class Rich:
    def __init__(self, name="singsiuk", budget="100 billion $"):
        self.name = name
        self.budget = budget

    # Getter Setter 생성
    def getName(self):
        print("name 의 getter 호출 ")
        return self.__name

    def getBudget(self):
        print("budget 의 getter 호출 ")
        return self.__budget

    def setName(self, name):
        self.__name = name

    def setBudget(self, budget):
        self.__budget = budget

    # name 속성을 외부에서 호출하면 getName 과 setName method 가 호출 된다
    name = property(getName, setName)
    budget = property(getBudget, setBudget)

rich = Rich()
rich.name = "sing"
rich.budget = "1999999999999원"
print(rich.getBudget(), rich.getName())
```

1999999999999원 sing