푸딩 3주차 스터디

Python

1	함수
2	파일 읽기와 쓰기
3	객체 지향 프로그래밍

함수 이해하기

```
print("A님. 두 숫자를 입력하세요.")
num1 = int(input("정수 1 => "))
num2 = int(input("정수 2 => "))
hap = num1 + num2
print("결과 :", hap)

print("B님. 두 숫자를 입력하세요.")
num1 = int(input("정수1 => "))
num2 = int(input("정수2 => "))
hap = num1 + num2
print("결과 :", hap)

print("C님. 두 숫자를 입력하세요.")
num1 = int(input("정수 1 => "))
num2 = int(input("정수 2 => "))
hap = num1 + num2
print("결과 : ", hap)
```

- 같은 동작 반복
  -> 동일한 코드를 몇 번씩 작성해야 함.

함수 이해하기

```
def hapFunc() :
    num1 = int(input("정수1 ==> "))
    num2 = int(input("정수2 ==>"))
    hap = num1 + num2
    print("결과 :", hap)

print("A님. 두 숫자를 입력하세요.")
hapFunc()

print("B님. 두 숫자를 입력하세요.")
hapFunc()

print("C님. 두 숫자를 입력하세요.")
hapFunc()
```

- 함수를 이용하면 소스 코드도 짧아지며, 코드의 변경이나 유지보수가 쉬움 (함수 부분만 수정하면 OK)

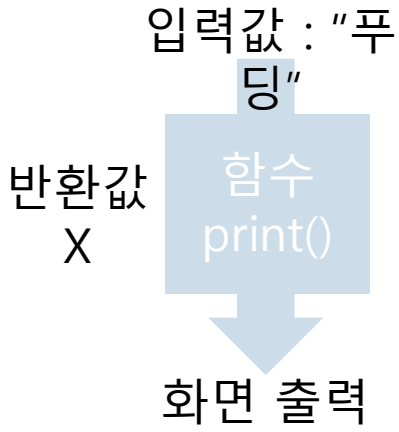
함수 이해하기

반환 값이 없는 함수

```
print("푸딩")
```

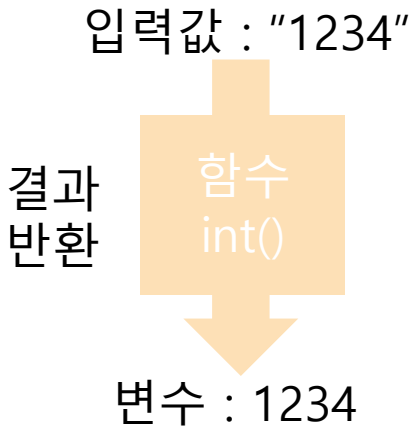
반환 값이 있는 함수

```
num = int("1234")
print(num)
```



- **반환 값이 없는 함수**

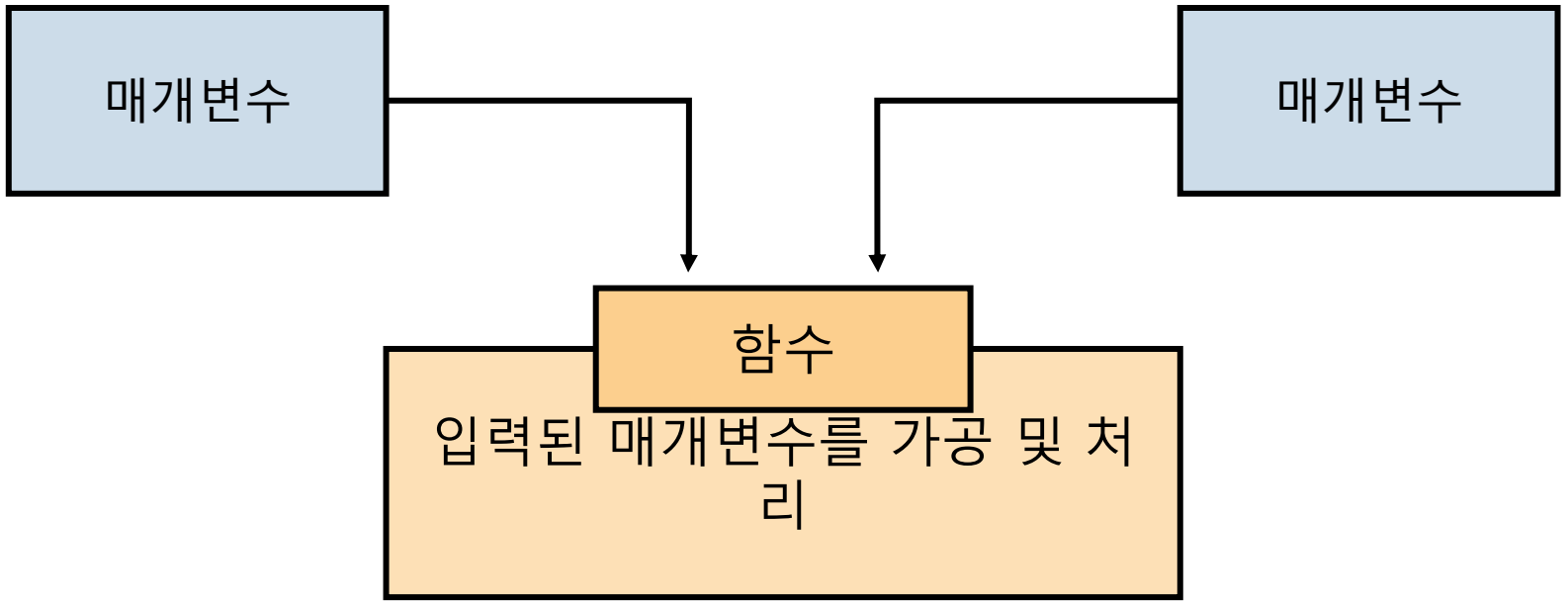
Ex) print()
: 입력받은 글자만 모니터에 출력하고,
별도로 돌려주는 값은 없음.



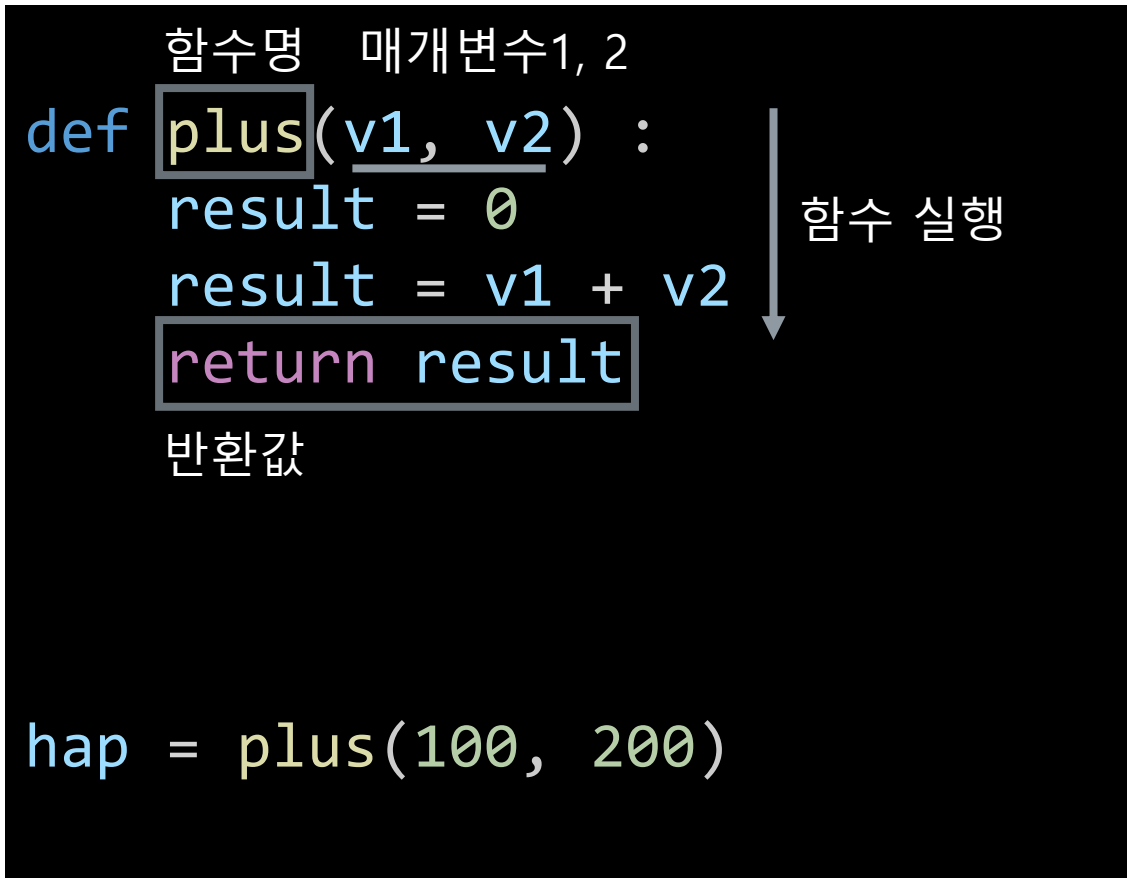
- **반환 값이 있는 함수**

Ex) int()
: 입력받은 문자열 -> 정수로 반환
반환하는 값을 받는 변수가 필요 (num)

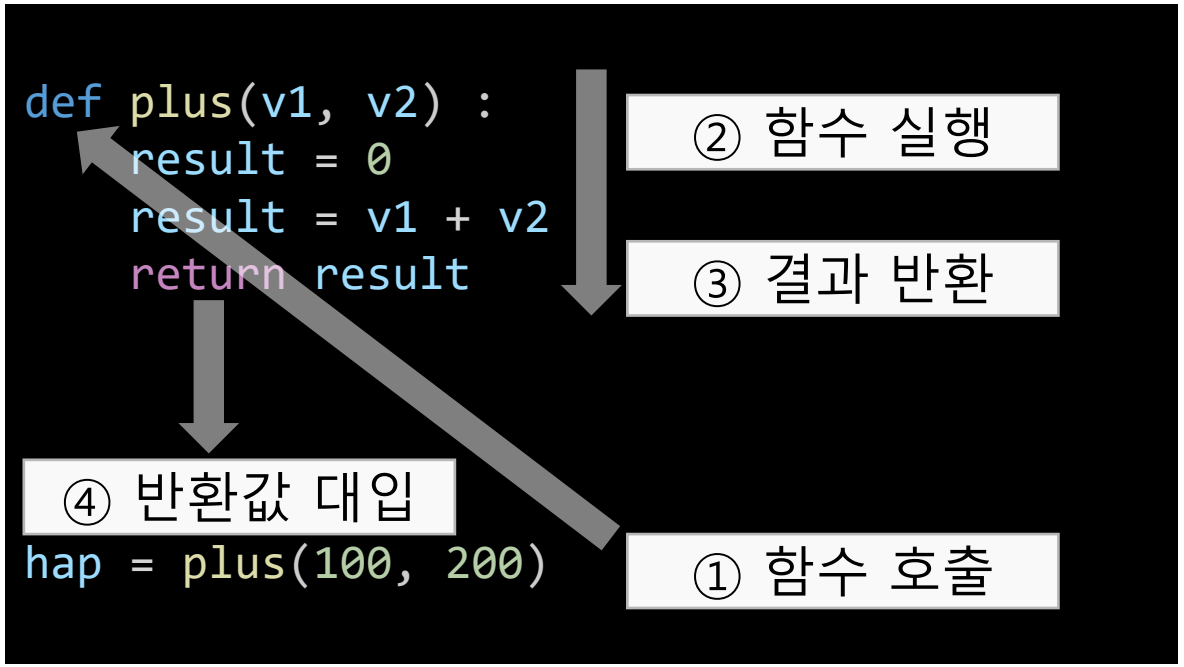
함수의 형태



함수의 형태와 호출 순서



함수의 형태와 호출 순서



- 함수 호출
- 함수 실행
- 결과 반환
- 변환값 대입

함수의 매개변수

```
def plus_2(v1, v2) :
    result = 0
    result = v1 + v2
    return result

hap = plus_2(100, 200)

def plus_3(v1, v2, v3) :
    result = 0
    result = v1 + v2 + v3
    return result

hap = plus_3(100, 200, 300)
```

▪ 매개변수의 개수
: 함수를 호출할 때는 정확히 매개변수의 개수에 맞춰서 호출해야 함.

-> plus_2에 매개변수 3개 : 오류
-> plus_3에 매개변수 2개 : 오류

매개변수가 없는 함수 생성 가능

함수의 매개변수

```
def plus_3(v1, v2, v3 = 0) :
    result = 0
    result = v1 + v2 + v3
    return result

hap = plus_3(100, 200)
```

▪ 매개변수의 기본값
: 매개변수란에 미리 설정

-> plus_3에 매개변수 2개 : 오류가 나지 않음
(v3값을 입력 받지 않아도 기본값이 0으로 설정되어 있으므로 오류 발생X)

함수의 반환값

```
def func1() :
    print("반환값이 없는 함수")

func1()

def func1() :
    print("반환값이 없는 함수")
    return

func1()
```

- 반환값이 없는 함수

return문 생략 가능
반환값 없이 return문만 써도 가능

함수의 반환값

```
def func2() :
    result = 100
    return result

x = func2()

def func3() :
    result1 = 100
    result2 = 200
    return result1, result2

x, y = func3()
```

- 반환값이 1개인 함수
: 반환값을 대입할 변수 1개 필요

- 반환값이 2개인 함수
: 반환값이 대입할 변수 2개 필요

pass

```
def myFunc() :
    pass

x = int(input("숫자 입력"))

if x % == 2 :
    pass
else :
    print("거짓입니다.")
```

▪ pass
: 아무것도 안한다는 의미
함수의 이름과 형태만 만들고 내부는 나중에 코딩하고 싶은 경우 등에 사용

- If 문이나 반복문에서도 아무것도 안하는 코드로 사용할 수 있음

지역변수와 전역변수

지역변수

한정된 지역(local)에서만 사용되는 변수

함수 1

A = 10
(-> 함수 안에서만 사용되는 지역변수)

A가 무엇인지 함수1에서 안다

함수2

A가 무엇인지 함수2에서 모른다

전역변수

프로그램 전체(global)에서 사용되는 변수

B = 20
(-> 프로그램 전체에서 사용되는 전역변수)

함수1

B가 무엇인지 함수2에서 안다

함수2

B가 무엇인지 함수2에서 안다

지역변수와 전역변수

```
def func1() :
    a = 10 # 지역변수
    print("func1()에서 a의 값", a)

def func2() :
    print("func2()에서 a의 값", a)

a= 20 # 전역변수

func1()
func2()
```

```
func1()에서 a의 값 10
func2()에서 a의 값 20
```

▪ 지역변수와 전역변수
:지역변수와 전역변수의 이름이 같은 경우, 지역변수가 우선됨.
func1() 실행 -> 지역변수 a = 10 우선
func2() 실행 -> 전역변수 a = 20 실행
(func2에는 a가 따로 설정X)

global

```
def func1() :
    global a
    a = 10 # 전역변수
    print("func1()에서 a의 값", a)

def func2() :
    print("func2()에서 a의 값", a)

a= 20 # 전역변수

func1()
func2()
```

▪ global
: 함수내에서, 지역변수 대신에 무조건 저녁변수로 사용하고 싶을 때 사용

func1() 안에서 a = 10을 전역변수로 지정함 ->
-> a = 20 변경
-> 하지만, 변경된 후에 함수를 호출하였으므로 다시 위로 올라가서 a = 10 이 됨
-> func1(), func2() 모두 10출력

파일 읽기

1단계
파일 열기

변수명 = open("파일 경로/파일 이름", "r")

- open() 함수에서 파일명 지정
- 읽기를 의미하는 "r"로 설정
- "r" 생략 가능 / "rt"라고 써도 가능

2단계
파일 읽기

readline() 함수 / readlines() 함수

3단계
파일 닫기

변수명.close()

한 행씩 읽기_readline()

```
inFile = None
inStr = ""

inFile =
open("C:/FirstPython/Chapter09/myDa
ta1.txt", "r", encoding = "UTF-8")

inStr = inFile.readline()
print(instr, end = '')

inStr = inFile.readline()
print(instr, end = '')

inStr = inFile.readline()
print(instr, end = '')

inFile.close()
```

▪ readline( )함수
: 한번에 한 행씩만 읽을 수 있음.
(100개의 행이 있다면 100번 반복해서 읽기)
-open( ) 함수
: 폴더 경로 설정 시 '/'(슬래시)로 \(백슬래시)X
: encoding = "UTF – 8" 한글 사용할 경우 필요

readline( ) 함수 3번 -> 3개의 행만 읽을 수 있음

(폴더 경로는 사용자에 따라 다름
위의 예시는 가상의 경로)

한 행씩 읽기_readline()

```
inFile = None
inStr = ""

inFile =
open("C:/FirstPython/Chapter09/myDa
ta1.txt", "r", encoding = "UTF-8")

While True :
    inStr = inFile.readline()
    if inStr == "":
        break
    print(inStr, end = '')

inFile.close()
```

▪ readline( )함수 이용해서 모든 행 읽기
-> While 함수 이용

: inStr == “” -> 읽어온 것이 없다면 break문 통해서 반복 빠져나옴

: 읽은 것이 있다면 읽은 내용 출력

한꺼번에 읽기_readlines( ) 함수

```
inFile = None
inList = []

inFile =
open("C:/FirstPython/Chapter09/myDa
ta1.txt", "r", encoding = "UTF-8")

inList = inFile.readlines()
print(inList)

inFile.close()
```

▪ readlines( )함수
: 파일의 내용을 한꺼번에 읽어서 '리스트'에 저장
-> 파일의 모든 행을 '리스트'로 저장해서 한번에 반환

한꺼번에 읽기_readlines( ) 함수

```
inFile = None
inList = []

inFile =
open("C:/FirstPython/Chapter09/myDat
a1.txt", "r", encoding = "UTF-8")

inList = inFile.readlines()
for inStr in inList :
    print(inStr, end='')

inFile.close()
```

▪ readlines( )함수 리스트 내용 한 줄씩 출력하기
: for문 이용해서 한 줄씩 출력

파일 쓰기

1단계
파일 열기

변수명 = open("파일 경로/파일 이름", "w")

- open() 함수에서 파일명 지정
- 쓰기를 의미하는 "w"로 설정

2단계
파일 쓰기

writelines() 함수 이용

3단계
파일 닫기

변수명.close()

파일 쓰기_writelines( ) 함수

```
outFile = None
outStr = ""

outFile =
open("C:/FirstPython/Chapter09/myDa
ta1.txt", "w")

outStr = "가나다라"
outFile.writelines(outStr + "\n")

outStr = "마바사아"
outFile.writelines(outStr + "\n")

outStr = "자차카타"
outFile.writelines(outStr + "\n")

outFile.close()
```

- writelines( )함수

다음줄로 넘어가기 위해 ”\n” 붙여주기

“\n” X -> 가나다라마바사아자차카타

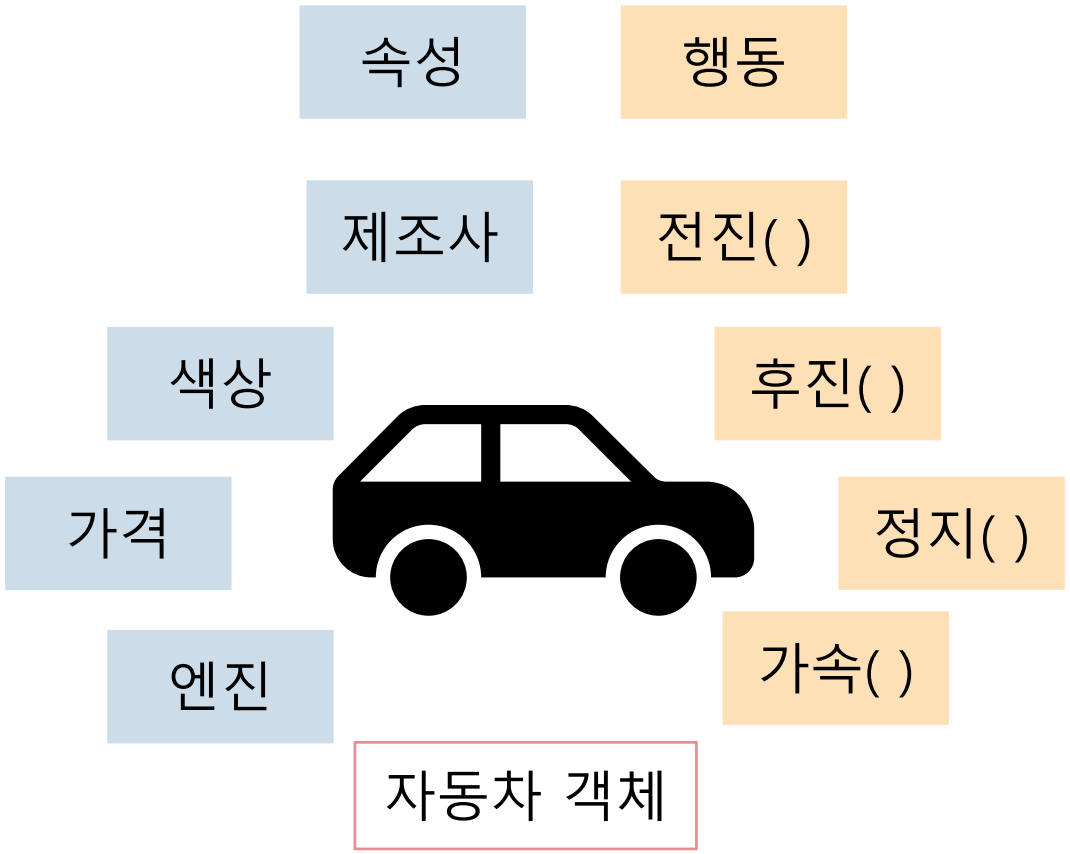
“\n” O -> 가나다라
마바사아
자차카타

파일 쓰기_writelines( ) 함수

```
outFile = None
outStr = ""

outFile =
open("C:/FirstPython/Chapter09/myData1.
txt", "w")

While True :
    outStr = input("내용 입력 ==>")
    if outStr != "" :
        outFile.writelines(outStr+"\n")
    else :
        break

outFile.close()
```

▪ 사용자에게 입력받은 내용을 파일에 쓰기
([Enter]누르면 파일 종료하도록)
While 함수 이용해서 사용자가 [Enter]누를 때까지 입력 받고, 그 내용을 파일에 씀.

객체 지향 프로그래밍

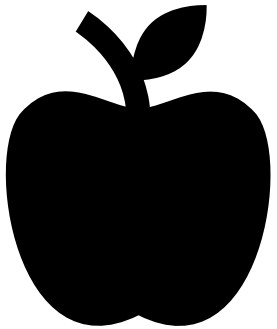


- 객체 지향 프로그래밍에서는 프로그램을 작성할 대상이 되는 실제 세계의 사물(객체)을 그대로 표현하고, 그것들이 어떻게 움직이는지 정해주고 그 객체들에게 일을 시킴.

- 객체 지향 프로그래밍을 잘 사용하면 좋은 프로그램을 빨리 만들 수 있고, 나중에 수정하기도 편함.

- 객체란 어떤 속성과 행동을 가지고 있는 데이터

클래스와 객체



사과 : 클래스

첫 번째 줄, 두번째에 있는 사과
: 객체(인스턴스)

클래스와 객체

```
class Rabbit :

    shape = ""
    x = 0
    y = 0

    def goto(self, x, y) :
        self.xPos = x
        self.yPos = y
```

클래스 생성
class 클래스 이름 :

속성 : 변수처럼 생성

행동 : 함수 형태로
단, 클래스 안에서 구현된 함수는 '메소드'라는 용어 사용

클래스에서 만든 메소드의 매개변수는
기본적으로 제일 앞에 self 써줌
(self.변수 -> 클래스 안에서 정의한 속성)

객체의 생성 & 속성 및 메소드

```
rabbit1 = Rabbit()
rabbit2 = Rabbit()

rabbit1.shape = "원"
rabbit2.shape = "삼각형"

rabbit1.goto(100, 100)
rabbit2.goto(-100, 100)
```

▪ 객체의 생성
(하나의 클래스에 여러 개의 객체 생성 가능)

▪ 속성에 값 대입하기
: '객체이름.속성이름' 으로 속성 값 대입 가능

▪ 메소드의 호출
: '객체이름.메소드이름'으로 호출 가능

(goto( ) : 이동하는 함수)

생성자

```
class Rabbit :
    shape = ""
    x = 0
    y = 0

    def __init__(self) :
        self.shape = '토끼'

    def goto(self, x, y) :
        self.xPos = x
        self.yPos = y

rabbit = Rabbit()
```

▪ 생성자란 객체를 생성하면 무조건 호출되는 메소드

▪ 생성자의 형태
: 클래스 안에서 __init__( )라는 이름으로 지정 (언더바 2개)

-> 실행결과 rabbit.shape 는 토끼로 자동 지정

매개변수가 있는 생성자

```
class Rabbit :
    shape = ""
    x = 0
    y = 0

    def __init__(self, value) :
        self.shape = value

    def goto(self, x, y) :
        self.xPos = x
        self.yPos = y

rabbit1 = Rabbit('원')
```

▪생성자에서 value라는 매개변소를 받도록
-> 넘겨받은 매개변수 값을 속성 중 shape에 대입

-> 객체를 생성하면서 shape을 지정하면
해당 모양으로 설정된 객체가 생성

소멸자

```
class Rabbit :
    shape = ''
    def __del__(self) :
        print("이제", self.shape,
"는 자유에요")

rabbit = Rabbit()
rabbit.shape = "도비"
del(rabbit)
```

```
이제 도비 는 자유에요
```

▪소멸자 __del__
: 객체가 제거될 때 자동으로 호출
(<-> 생성자 __init__)

객체 제거 :del(객체)

-> 이때 소멸자 호출

객체끼리 덧셈

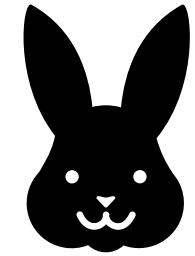
```
class Rabbit():
    shape = ""
    def __add__(self, other) :
        print("객체", self.shape,
"와", other.shape, "가 친구가 되었
습니다.")

rabbit1 = Rabbit()
rabbit1.shape = "토끼"

rabbit2 = Rabbit()
rabbit2.shape = "도비"

rabbit1 + rabbit2
```

```
객체 토끼 와 도비 가 친구가 되었습니다.
```

- __add__ ( )
: 객체끼리의 덧셈을 할 경우에 실행
self.shape -> 나의 모양 : rabbit1
other.shape -> 다른 것의 모양 : rabbit2

클래스의 상속

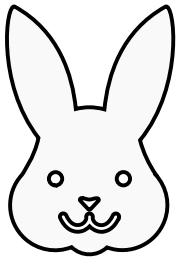
슈퍼 클래스
(부모 클래스)



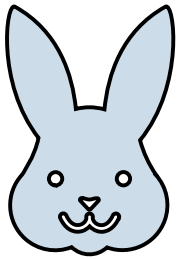
상속

상속

class 토끼
속성 : 모양, x,y
행동 : 이동하기()



class 집토끼
속성 : 토끼 속성 + 주인이름
행동 : 토끼 행동 + 사료먹기()



class 산토끼
속성 : 토끼 속성 + 산이름
행동 : 토끼 행동 + 들풀먹기()

서브 클래스
(자식 클래스)

토끼 클래스의 속성과 행동을 그대로 이어받고
추가적인 속성과 메소드를 추가

클래스의 상속

```
class Rabbit :
    shape = ""
    xPos = 0
    yPos = 0
    def goto(self, x, y) :
        self.xPos = x
        self.yPos = y

class HouseRabbit(Rabbit) :
    owner = ""
    def eatFeed() :
        print("집토끼가 사료를 먹습니다")

class MountainRabbit(Rabbit) :
    mountain = ""
    def eatWildgrass() :
        print("산토끼가 들풀을 먹습니다")
```

- 서브 클래스 코드 구현
  - class 서브 클래스(슈퍼 클래스) :

토끼 클래스를 상속받은 집토끼 클래스 정의

토끼 클래스를 상속받은 산토끼 클래스 정의

클래스의 상속

```
hRabbit = HouseRabbit()
mRabbit = MountainRabbit()

hRabbit.shape = '원'

mRabbit.goto(100, 100)
```

▪서브 클래스의 객체 생성, 속성 값 대입,
메소드 호출 일반 클래스와 동일

HouseRabbit 클래스에 존재하지 않는 shape 속성
-> 슈퍼 클래스인 Rabbit에 정의되어 있기 때문에 사용 가능

MountainRabbit 클래스에 존재하지 않는 goto 메소드
-> 슈퍼 클래스인 Rabbit에 지정되어 있기 때문에 사용 가능

⇒ 서브 클래스는 슈퍼 클래스의 모든 속성과 행동 사용 가능

☺ 수고하셨습니다 ☺