

## 1) Python 이란 무엇인가?

인터프리터언어.

Python 프로그램은 공통적으로 유지보수하기 쉽다.

시스템 프로그래밍이나 하드웨어 제어와 같은 매우 특장하고 복잡영성이 많은 경우에는 적합!

→ 보완: 다른 언어로 만든 프로그램을 파일로 프로그램에 포함시킬 수 있다.

파이썬과 C는 칠레공동체다.

프로그램의 간단한 뼈대는 파이썬으로 만들고,

비록 설계도와 필터는 부분은 C로 만들어 파이썬 프로그램에 포함시킬 수 있다.

파이썬 라이브러리 중에는 순수파이썬으로만 저작된 것도 많지만.

C로 만든 것도 많다.

파이썬은 출처를 막추지 않으면 실행되지 않는다.

파이썬으로 무엇을 할 수 있을까?

1) 할 수 있는 일  
① 시스템 유틸리티 제작

② GUI 프로그래밍

③ C/C++ 와의 결합

④ 웹 프로그래밍

⑤ 소才是真正 프로그래밍

⑥ 데이터 분석 / IOT

2) 할 수 없는 일

① 시스템과 링크한 프로그래밍 영역

② 모바일 프로그래밍

## ◦ 시스템 유틸리티 제작

: OS의 시스템 명령어를 사용할 수 있는 작동도구를 갖추고 싶기 때문에 이를 바탕으로 할 가지 시스템 유틸리티 만드는데 유리

## ◦ GUI 프로그래밍

: Tkinter를 사용해GUI의 코드만으로 기능을 펴울 수 있음.

## ◦ C / C++ 코딩 결합.

: 과정성은 접촉 인터페이스로 부르는데, 그 이유는 다른언어 잘 아는  
개발자 사용할 수 있기 때문

C/C++ 프로그램  $\longleftrightarrow$  Python 프로그램

사용 가능.

## ◦ 웹프로그래밍

: 장고 프레임워크

## ◦ 수학 연산 프로그래밍

: NumPy라는 수학 연산 모듈 제공 (C로 적어둠)

## ◦ DB 프로그래밍

: Sybase, Informix, Oracle, MySQL, PostgreSQL 등  
DB 접근 도구 제공

pickle 블록: 파일이나 사용하는 자료 변형 없이 그대로

파일에 저장하고 불러오는 것은 말해 합지.

## ◦ 데이터 분석, IoT

: Pandas 모듈 사용하면 데이터 분석을 더 쉽고 효과적으로 할 수 있음.

리在外面포터 제어 도구로 사용 가능.

## ◦ 시스템과 통합한 프로그래밍 영역

: OS나 앱하는 후에는 반복적인 연산이 필요한 프로그램

데이터 앱을 만들거나 개발 프로그램을 만드는 것은 어렵다.

마우스 빠른 속도로 요구하거나 GUI를 적용한다는 프로그램에는 적합 >

## ◦ 모바일 프로그래밍

: 앱 개발 (Android, iOS)은 아직 어렵다.

안드로이드이나 표준 표준이 설정되어 지원은 있지만,

어려운 유통망은 앱 만들기는 여부를.

언어파티션?: 사용자가 입력한 소스코드를 실행하는 툴

\* 파이썬 재현형 언어프로세서 Python shell 이라고도 부름.

>>> 는 프롬프트라고 한다.

종료

단축키

>> import sys

Ctrl + Z → Enter

>> sys.exit()

기호문법 풀려오기

## 1) 산술연산

더하기	나누기 / 곱셈 /
>>> 1 + 2	>>> 3 * 9
3	27

## 2) 변수와 문자열 예선.

>>> a = 1

>>> b = 2

>>> a + b

3

## 3) 변수에 문자저장하고 출력

>>> a = "python"

>>> print(a)

python

## 4) 조건문 if

>>> a = 3

>>> if a > 1:

... print("a is greater than 1")  
... top for spacebar 4번으로 줄여쓰기 해야죠 (안된다면 키보드)  
아직 문장이 끝나지 않았음을 의미.

## 5) 반복문 for

>>> for a in [1, 2, 3]:

... print(a)

... `while` `enter`에서 끝남을 알려줘야.

## 6) 반복문 `while`

```
>>> i = 0  
>>> while T < 3:  
...     T = i + 1  
...     print(i)
```

7) 함수  keyword.

```
>>> def add(a, b):  
...     return a+b  
>>> add(3, 4)
```

## 예외 사용

: 대회형 인터프리터는 간접한 예제를 풀 때는 편리.

한계: 여러 줄의 복잡한 소스코드를 가진 프로그램을 만들 때는 불편.

또, 인터프리터 종료할 때마다 프로그램이 시작하기 때문에  
다시 실행해지게 되는 단점.

그냥 어떤 사용하기 위한 프로그램 만들 때는 예외 사용

# : 한글자리 주석

visual code → \*.py

'''  
'''  
''' } 여러줄 주석  
'''

터미널에서 컴파일

python \*.py

## 2) Python 프로그래밍 기초, 자료형

### 1) 숫자형

: 숫자형으로 이루어진 자료형.

• 정수형 - 양, 음, 0

• 실수형 - 소수점, 유포기 ( $4.24 \times 10^{10} = 4.24e10 = 4.24E10$ )

• 8진수 16진수 - 0o177 (8진수)  
0xABC (16진수)

$x^y \Rightarrow x**y$

$x//y \Rightarrow x$ 를  $y$ 로 나눈 몫

$x \% y \Rightarrow x$ 를  $y$ 로 나눈 나머지

### 2) 문자열

: 문자, 문자열로 구성된 문자들의 집합

만드는 법 4가지

" "

' '

''' '''

.... ....

en 4가지나 있을까?

1) 문자열에 '를 포함시키고 싶을 때

" " " 조합방법 \\'

2) .. " 를 .. ' 를 써도된다.

' ' ' 조합방법 \\"



$\text{str}[-\infty] = \text{str}[0]$

$\text{str}[-2]$  : 원래는 두번째

$\text{str}[-3]$  : 원래는 세번째

slicing : 특정 문자 잘라내기

$a[0:4] \rightarrow 0 \leq <4$

0 ~ 3의 범위.

$a[0:]$

0번 끝까지

$a[:10]$

처음부터 10까지.

$a[:]$

전체

$a[10:-7]$  가능

슬라이싱으로 문자를 나누기

$a = "20010331 Rainy"$

$\text{date} = a[:8]$

$\text{weather} = a[8:]$

$\text{date} = 20010331$

}  
자주 사용되는  
기법

Weather - Rainy ↴

문자열은 immutable 한가지

내부적으로 강제 번경 불가

시작이나 끝을 바꾸는 것은

a = "Python"

>>> a[:-1]

'P'

>>> a[2:]

"thon"

>>> a[:-1] + 'y' + a[2:]

다른식으로 출력

## 문자열 포맷팅

1) 숫자 바로 대입.

>> "I eat %d apples." %3

2) 문자열 바로 대입.

>> "I eat %s apples." %r five

3) 숫자 값을 나타내는 변수로 대입.

>> number = 3

>> "I eat %d apples." % number

4) 2개 이상의 값 넣기.

>> number = 10

>> day = "three"

>>> "I ate %d apples. So I was sick for %s days." % (number, day)

### \* Notice

- %.s 는 어떤 형태의 값이든 뜻한 문자열로 변환되어 들어감.
- %% 는 % 문자 자체를 설정.

그것으로 숫자 함께 사용.

- %.10s 전체 문자가 10자인 문자열로  
앞쪽 문자는 빙글거린 공백.
- %.10s 완쪽정렬.
- %.10.4f → 10자인 자리고 소수점 네 번째자리면 나온다.

format 함수를 사용한 포맷팅

숫자바꾸기

"I eat {0} apples.". format(3)

문자열바꾸기

"I eat {0} apples.". format("five")

숫자 값을 가진 변수로 바꾸기

number = 3

"I eat {0} apples.". format(number)

2개 이상의 값 넣기

number = 10

day = "three"

```
>>> "I ate {0} apples. So I was sick for {1} days.".format(10, 3)
```

이동으로

원복 정법

"{0:<10}.format("hi")

사이즈 10 흰색상자 원형판

한국 고전

```
"{0:>10} .format("hi")
```

가을내성금

```
"{} o: ^163 . format ("hi")
```

공백을 끌어온다로

```
"FO:三^10}.format("hi")
```

== == hi == == 아기치거나 노년증상.

소수점 표현하기

```
>>> "{0: 10.4f}".format(y)      default은 6자리.
```

16자리 입체 소스집 저작권자.

{ } 또는 문자표현합니다

```
>>> "{{and}}".format()
```

'{and}'

Python 워크숍 (Python 3.6↑)

문자열 앞의  $f$  접두어를 붙이면  $f$  문자열 조매핑 가능성을 사용할 수 있다.





### a. strip()

앞쪽 공백 제거는 lstrip()

뒤쪽 공백 제거는 rstrip()

### 8) 문자열 바꾸기 (replace)

>>> "Life is too short"

a.replace("Life", "Your leg")

Your leg is too short

### 9) 문자열 나누기 (split)

>>> a = "Life is too short"

a.split()

↑ 구분자를 명시하지 않으면 공백, 탭, 개행 기호로 나눠짐.

a = "a,b,c,d"

tokens라는 함수.

a.split(',')

### 3) 리스트 자료형

리스트를 만들 때는, [ ] 를 감싸고 , 로 구분시킨다.

odd = [1, 3, 5, 7, 9] 이런식으로.

[ ] 와 같이 빈 리스트 가능하고 표로 문자열, 숫자, 초기, 리스트 자체를 가질 수도 있고

어떤 자료형이든 들어가 될 수 있다.

빈리스트 예제 a = list()

리스트의 인덱싱과 슬라이싱.

인덱싱 a[i] ; 번째 원소

a[-1] 마지막 a[-2]

a[:-1] : 마지막 원소 제외.

$\alpha = [1, 2, ['a', 'b', 'c']]$

'b'를 끄집어내는 법  $\alpha[-1][1]$

1중 리스트라도  $\alpha[ ][ ]$

n개 써서 참조하면되는데 흔히로 인해 자주 쓰는 방식은 x

### 리스트의 슬라이싱

$\alpha[0:2]$

중첩리스트 슬라이싱

{  $\alpha[:i]$  ; 기준으로.  
  | 2개 3개 4개  
  |  $\alpha[i:]$  ← i는 여기도 가능.

$\alpha = [1, 2, 3, ['a', 'b', 'c'], 4, 5]$

$\alpha[3][:2]$   
 $\Rightarrow ['a', 'b']$

### 리스트 연산

1) 합집합 (+)

$\alpha = [1, 2]$

$b = [3, 4]$

$\alpha + b = [1, 2, 3, 4]$

2) 팩 (\*.\*)

$\alpha = [1, 2]$

$\alpha * 2$

$\alpha = [1, 2, 1, 2]$

3) 길이 구하기 (len())

$\alpha = [1, 2]$

$len(\alpha)$

$str()$  → 정수나 실수를 문자열로

=2

### 리스트 수정과 삭제

1) 수정

$\alpha = [1, 2, 3]$

$\alpha[2] = 4$  가능 : 문자열과의 차이.

## 2) 삭제

`del a[1]`

$a = [1, 4]$

`del a[x]`

`del` 개체

파이썬에서 사용하는 모든 자료형.

`del a[2:]` 여러개 삭제 가능.

리스트 관련 함수들.  $a = [1, 2, 3, 4]$

### 1) 리스트에 값 추가 (append)

$a.append(5) \rightarrow$  맨 마지막에 5를 붙임.

$a \Rightarrow [1, 2, 3, 4, 5]$

$a.append([4, 5]) \rightarrow$  맨 마지막에 [4, 5]라는 리스트 추가.

$a \Rightarrow [1, 2, 3, 4, 5, [4, 5]]$

### 2) 리스트 정렬 (sort)

값의 자료형이 같은때만 사용 가능

`del a[6]`

$a.sort()$  리스트는 사전 순 길이 5으로 정렬

$a \Rightarrow [1, 2, 3, 4, 5]$

### 3) 리스트 뒤집기 (reverse)

$a.reverse()$

$a \Rightarrow [5, 4, 3, 2, 1]$  처음 자료형이나.

### 4) 위치 반환 (index)

$a.index(3) \Rightarrow 2$

$a.index(6) \Rightarrow \text{Error}$  0이 없으므로

5) 노드 삽입 (insert)  $\Leftrightarrow$  append는 끝에 맨다.

5) insert ( $a, b$ )  $a$ 번째에  $b$ 를 삽입.

$a.insert(0, \underline{6})$  ↗ 여기에도 어느자리든 노 matter.

$a \Rightarrow [6, 5, 4, 3, 2, 1]$

6) 리스트 요소 제거 (remove)

$a.remove(4) \rightarrow$  첫번째 나오는 4를 제거.

$a \Rightarrow [6, 5, 3, 2, 1]$

7) 리스트 요소 끊어내기 (pop)

리스트의 맨 마지막 요소를 뺀다. 그 요소 삭제

$a.pop() \Rightarrow 1$

$a \Rightarrow [6, 5, 3, 2]$

$a.pop(2) \rightarrow$  2번째 요소 삭제 후 반환.  $\Rightarrow 3$

$a \Rightarrow [6, 5, 2]$

8) 리스트에 포함된 요소  $x$ 의 개수 세기  $\Rightarrow$  전체 개수는  $len(a)$

$a.count(x) \rightarrow x$ 가 리스트에 몇개 있는지?

$a.count(5) \Rightarrow 1$

9) 리스트 확장 (extend)  $\Rightarrow$  += 과 동일

$a.extend([1, 2])$

↑ 이자리에는 반드시 리스트만.

$a \Rightarrow [6, 5, 2, 1, 2]$

3) 튜플 자료형

몇 가지 점을 제외하고 리스트와 거의 비슷

리스트와 차이

o 리스트는 []로 둘러싸지만, 튜플은 ()로 둘러싼다.

o 리스트는 그 값의 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 바꿀 수 없다.

## 튜플 예시

t1 = ()

t2 = (1,) → 1개의 요소를 가질 때 오로 텐데 반드시 , 를 붙여야 함.

t3 = (1, 2, 3)

t4 = 1, 2, 3 → 괄호 생략 가능.

t5 = ('a', 'b', ('ab', 'cd'))

튜플과 리스트는 비슷한 역할을 하지만, 프로그래밍 시에 튜플과 리스트를 구별해 사용하는 것이 유익하다.

둘의 가장 큰 차이는 값을 '변경시킬 수 있는가?' 의 여부이다.

프로그래밍에서 흔히가는 종종 값이 고정되거나 때는 튜플을 사용  
하지만 리스트를 사용하도록 한다.

값을 바꾸거나 삭제시켜야 하면 에기 방식.

## 튜플 다루기.

### 1) 인덱싱

t[x] → x 번째 요소.

t[x:] → x부터 끝까지.

### 2) 더하기

t1 = 1, 2

t2 = 3, 4

t1 + t2

(1, 2, 3, 4)

### 3) 슬라이싱

$t1 = 1, 2$

$t1 \neq 2$

$(1, 2, 1, 2)$

4) 길이 구하기

$\text{len}(t1)$

4

5) 딕셔너리 자료형.

딕셔너리란? key-value를 한 쌍으로 갖는 자료형

Hash 또는 연관 배열이라고 함.

딕셔너리는 리스트나 튜플처럼 순차적으로 저장된 값을 구하지 않고 key를 통해 value를 얻는다.

이것이 가장 큰 특징이다. 단어 뜻을 찾기 위해 사전의 내용을 순차적으로 모두 검색하는 것과  
아니라 찾는 단어가 있는 곳만 펼쳐보는 것.

딕셔너리 생성법

$dict = \{ 'name': 'Pey', 'phone': '010-4443-221' \}$

$\begin{matrix} \text{key}_1 = \text{value}_1, & \text{key}_2 = \text{value}_2, \dots \end{matrix}$  이런 형식.  
 $\uparrow$  어느 자료형 이런 생략

주의점 : key는 번호와 같은 값을 사용, value에는 번호와 같은 값과 번호를 갖 모두 사용 가능

$a = \{ 1: [1, 2] \}$

딕셔너리 상 추가, 삭제

1) 추가.

현재  $a = \{ 1: 'a' \}$

추가  $a[2] = 'b'$

$a[3] = [1, 2, 3]$

$\alpha$  [key] = value  
 $\alpha = \{1: 'a', 2: 'b'\}$

$\alpha = \{ 1: 'a', 2: 'b', 3: [1, 2, 3] \}$

☞  $\Delta[(1,2)] = [1,2,3]$  를까? xs

$\Delta[[1,2]] = [1,2,3]$  될까? No

why? list는 변환 가능한 Data-type 이고,  
이것은 key로 부적합.

## 2) 삭제

def a[key]

## 딕셔너리 사용법.

어디에 사용하는 것이 좋을까?

유익한 데이터의 특성을 나타내며 좋다. ex) 학생들의 성적.

작장인들의 연봉 .

key를 사용해 value 얻기.

리스트나 튜플과 달리 인덱싱과 슬라이싱이 가능합니다.

앞의 단 하나의 방법뿐인가? 이것은, key를 이용하여 value를 구하는 것이다.

ə [key]

\* 주의사항 : key는 고유한 값이므로 중복되는 key를 설정해 놓으면, 하나를 제외하면 나머지 모두 무시

$$A = \{ 1: 'a', 1: 'b', 1: 'c' \}$$

0

138

이유는 key를 통해 value를 얻는 데,

key가 정해지지 않으면 어떤 key에 해당하는 value를 불러야 하는지  
ambigious 이다.

딕셔너리 key 를 넘겨줫을 때 리스트로 리턴해 주는 틱스 .  
오류 발생 .

딕셔너리 관련 험수들 .

1) key 리스트 만들기 (keys) : 딕셔너리의 키만 모아 dict\_keys 형태로 반환 .

a = { 'name' : 'pey' , 'phone' : '0109943323' , 'birth' : '1988' }

a.keys()

dict\_keys(['name', 'phone', 'birth'])

\* 참고 python 3.0 keys() 함수 \*

python 2.7 까지는 dict\_keys() 객체가 아니라 리스트를 반환했었는데,

리스트를 반환시켜야할 때 딕셔너리 냥비가 발생 . 3.0 이후부터는

이러한 디버깅 냥비를 줄여기 위해 dict\_keys() 객체를 반환시킨다 .

리스트를 반환 받으려면, list(a.keys())를 사용한다 .

dict\_values() , dict\_items 역시 3.0 이후부터 추가된 것들 .

리스트로 변환되지 않더라도 . 반복문 실행 가능하도록 설계됨 .

for k in a.keys():

print(k) dict\_keys() 객체는 리스트 사용과 차이가 있으니 .

append, insert, pop, remove, sort는 사용불가 .

2) Value 리스트 만들기 (values)

a.values()

dict\_values() 객체가 반환된다 .

3) key, value 쌍 묵기 (items)

a.items()

dict\_items([('name', 'pey') , ... ])

key, value 을 튜플로 묶은 것을 dict\_items 객체로 반환 .

#### 4) key, value 삽입하기 (clear)

a.clear() 딕셔너리 안의 모든 요소 삭제

$a \Rightarrow \{\}$

#### 5) key로 value 얻기 (get)

a.get(key)  $a[key]$  허용

value. key가 없는 경우  $a[key]$ 는 예외

a.get(key) 는 None을 반환.

딕셔너리 안에 찾으려는 key값이 없을 경우 예외를 발생하는 대신 가려오지 않고 None을 반환.

get(k, 디폴트값) 을 사용하자.

#### 6) 해당 key가 딕셔너리에 몇갠지 있는지 조사.(in)

key in <dic-name>

예시 'name' in a

True or False 리턴.

#### 6) 집합자료형

집합에 포함된 것을 쉽게 처리하기 위해 만든 자료형.

Set 구조를 통해 만들고.

s1 = set([1,2,3])

$s1 \Rightarrow \{1, 2, 3\}$  \* s = set() empty set.

s1 = set("Hello")

$s1 \Rightarrow \{'e', 'H', 'l', 'o'\}$

↳ 1개 문자 빼고 순서가 바뀌어도됨.

why? Set의 특성.

◦ 중복을 허용하지 X

◦ 순서가 없다.

리스트나 튜플은 순서가 있지만, 인덱싱을 통해 자료형의 값을 얻을 수 있지만 `set` 자료형은 순서가 없애 인덱싱 불가.

만약, *yet* 자료형에서 *저장된* 값을 *연속성으로* 접근하여연 *마음과 같이* *리스트나 퍼플*로 *변환*

\* 중복을 허용하지 않는 set의 특징은 자료형의 중복을 제거하기 위한 필수 역할로 종종 사용 \*

$S1 = \text{Set}([1, 2, 3])$

`t1 = tuple(s1)`

[1, 2, 3] t1

|1[0] (1,2,3)

$$1 + 1[0] = 1.$$

교집합, 합집합, 차집합.

```
S1 = set ([1,2,3,4,5,6])
```

$$S2 = \{4, 5, 6, 7, 8, 9\}$$

### 1) 고장난

$$\textcircled{1} \quad S1 \setminus S2 \Rightarrow \{4,5,6\}$$

② §1. Intersection (s2)  $\Rightarrow \{4, 5, 6\}$

## 2) 향자합.

①  $\$1 + \$2 \Rightarrow \{[1, 2, 3, 4, 5, 6, 7, 8, 9]\}$

④ `s1.Union(s2)` ⇒ { [1, 2, 3, 4, 5, 6, 7, 8, 9] }

### 3) 차집합.

Ø J1 - S2  $\Rightarrow \{1, 2, 3\}$

## ② S1. difference (S2)

나라면 짜놓았지라.

Set 자료형 관련 험수들.

1) 값 1개 추가 (add)

s1.add(7)

2) 값 여러개 추가 (update)

s1.update([1,2,3])

3) 특정값 제거

s1.remove(2)

4) Bool 자료형.

참과 거짓 2가지 종만 가능수다.

\* 주의 예상이 True, False임.

true, false로 쓰면 오류 발생.

type(a) → a의 자료형을 반환.

사용되는 3.

제일의 반복문

자료형에도 참과 거짓이 있다.

자료가 비어있을 때 → 거짓 아니면 참

" () [] {} 들은 거짓.

이걸 어디에 쓸까?

while a:  
print(a.pop(0))

a가 비어있거나 pop.

자료형의 참, 거짓 판斷

if []:

  ~

bool(0)

else

  ~ ✓

bool([]) → False

bool([1, 2]) → True

8) 자료형의 값을 저장하는 공간, 변수.

C나 Java는 변수 생성시, 자료형을 적정 지정해와 파이썬은

저장값을 스스로 판斷하여 자료형을 적용

변수란?

파이썬에서 사용되는 변수는 가변성을 지니는 것이라고 말할 수 있다.

a = [1, 2, 3]

[1, 2, 3]의 값을 가지는 리스트 자료형(객체)가 차등으로 메모리에 생성

a는 [1, 2, 3] 리스트를 저장된 메모리의 주소를 가리킴.

id(a) 를 주목할 수 있다.

리스트를 복사하고자 할 때.

a = [1, 2, 3]

b = a

id(a) == id(b) 를 해보면 같은 주소나온다. 동일주소 가리기인가 예상.

a is b True a와 b가 동일 객체인지 검사하는 IS 연산.

a[2] = 4

[1, 2, 4]

↑  
a  
b      b도 값이 바뀜.

그렇다면 빠른 주소를 서로 다른 주소를 가리키게 하는 법.

① [:] 이용.

a = [1, 2, 3]

$b = a[:]$

② copy 모듈 이용

변수 또는 클리스를 모아놓은 파일

다른 파일에서 프로그램에 불러와 사용할 수 있게끔 만든 파일.

사용법.

>>> from copy import copy

>>> b = copy(a)

변수를 만드는 여러가지 방법.

리스트를 변수

a, b = ('Python', 'life')

(a, b) = 'Python', 'life'

↑  
a  
↑  
b

[a, b] = ['Python', 'life']

a = b = 'Python'

swap 기술

a = 3

b = 5

a, b = b, a ✓ swap 기술.

### 3) 프로그램의 구조를 알아가기 위한 기본구조.

If문의 기본구조.

들여쓰기 (Indentation)

If 조건문 :

↳ 들여쓰기 해지 않으면 오류

    > 언제나 같은 네비로 들여쓰기가 되어야함. (space 4자 or Tab)

else :

들어오는 : (줄) 허가되지 않도록 주의.

~~~~

### 1) or, and, not 연산

예시 1) If money >= 2000 or card :

{ If 금액은 (and) B교시 :

수업있음.

| If not money:

2) in, not in

X in 리스트

X not in 리스트

X in 투플

X not in 투플

X in 딕셔너리

X not in 딕셔너리

아무일도 일어나지 하지 않고 있을 때.

If 조건:

pass

else:

—

If ~ elif ~ else

한동작일 때 if 'money' in pocket: pass 같은 끝이 위치 가능.  
else: print("카드를 깨워라")

조건부 표현식 (?: 연산자)

조금이라 차이 허용 If 조건을 else 조건에 거짓인 경우

message = "success" if Score >= 60 else "failure"

while 문

while 조건문:

— —  
final

문제

강제로 빠져나가기 break

맨 처음으로 돌아가기 continue

무한루프 while True :

Ctrl+C → 강제 종료.

for 문

기본구조.

for 변수 in 리스트(튜플, 문자열) :

    ~~~~~  
    돌파선기 ~~~~~

1) 전형적인 for 문.

```
test_list = ['one', 'two', 'three']
```

```
for i in test_list:
```

```
    print(i)
```

2) 다중원 for 문 사용.

```
a = [(1,2), (3,4), (5,6)]
```

for (first, last) in a:     튜플의 변수를 사용.

```
    print(first+last)
```

3) continue 건너뛰기 in.

4) for 문과 함께 자주 사용되는 range 함수

숫자리스트를 자동으로 만들어 주는 range 함수와 함께 사용하는 경우가 많다.

```
a = range(10)
```

a

```
range(0,10)
```

range (시작점, 끝점)      range (끝점)  
add = 0                          ← 디폴트

```
for i in range(1, 11)
    add = add + i
print(add)
```

자주쓰이는 형태.

```
marks = [90, 25, 67, 45, 80]
for number in range(len(marks)):
    if marks[number] < 60:
        continue
    print("7.1번 문제 축하합니다. 합격입니다." + (number + 1))
```

1개 for문

구간

```
for i in range(2, 10)
    for j in range(1, 10)
        print(i*j, end=" ") → 공백으로 파악
print()
```

리스트 내포 사용하기 (List Comprehension)

이전 예시가 떠오르다

$\alpha = [1, 2, 3, 4]$

result = []

for i in  $\alpha$ :

result.append(i \* 3)

print(result)

[3, 6, 9, 12]

$\alpha = [1, 2, 3, 4]$

result = [num \* 3 for num in  $\alpha$ ]

print(result)

[3, 6, 9, 12]

만약 1, 2, 3, 4 중 짝수에만 3을 곱하고 싶다. (제곱사용 가능)

`result = [num * 3 for num in A if num % 2 == 0]`

[표현식 for 항목 in 반복가능 객체 if 조건]

일반화

[표현식 for 항목1 in 반복가능 객체1 if 조건1

for 항목2 in 반복가능 객체2 if 조건2

:

for 항목n in 반복가능 객체n if 조건n]

#### 4) 프로그램의 입력과 출력은 어떻게?

함수

1) 함수를 사용하는 이유?

반복적으로 사용되는 가치 있는 부분을 한 끓치로 묶어서

“어떤 입력값을 주었을 때 어떤 결과값을 출력한다”

프로그램의 흐름을 일목요연하게 볼 수 있다.

입력값이 여러 함수를 거쳐면서 결괏값을 내는 것을 한 눈에 보는 것.

오류발생 위치, 프로그램의 흐름을 잡기 쉽다.

2) 함수의 구조.

`def <함수명> (<매개변수>) :`

예시      `def add(a, b) :`

~~~~~

~~~~~ return a+b

~~~~~

~~~~~ 들여쓰기

~~~~~

매개변수와 인수 (Parameter and Argument)

매개변수는 함수에 입력으로 전달된 값을 받는 변수

인수는 함수로 호출할 때 전달하는 입력값.

def add(a, b): *매개변수*

return a+b

C = add(3, 4) *→ 인수*

### 3) 입력값과 결과값에 따른 함수의 형태

함수의 형태는 입력값과 결과값의 종류 유무에 따라 4가지 유형으로 나온다.

① 일반적인 함수 : 입력값 O, 결과값 O

def 함수(매개변수):

return 결과값

② 입력값이 없는 함수

def 함수():

return 'Hi'

③ 결과값이 없는 함수

결과값은 오직 return문일 뿐 출력받을 수 있음.

def 함수(매개변수):

\_\_\_\_\_

def add(a, b):

print("x.d&y.d의 합은 %d 입니다." % (a, b, a+b))

a = add(3, 4)

print(a) → None

④ 입력값, 결과값 모두 없는 함수

def say():

print('Hi')

#### 4) 매개변수 지정하여 호출하기

```
def add(a,b):
```

```
    return a+b
```

```
result = add(a=3, b=4) > 순서상은 X
```

```
result = add(b=5, a=3)    장점: 순서와 관계 X
```

#### 5) 가변인자.

```
def 할수이름 (*매개변수) *
```

예시

```
def add_many(*args):
```

```
    result = 0      매개변수 이름앞에 *를 붙이면 .
```

```
    for i in args      입력값을 전부 모아서 튜플로 만든다.
```

```
        result += i      add_many(1,2,3,4,5)
```

```
    return result      By args는 (1,2,3,4,5) 가 된다.
```

#### 키워드 파라미터 kwargs (keyword arguments)

사용시 앞에 \*\*를 붙인다.

```
def print_kwargs(**kwargs):
```

```
    print(kwargs)
```

사용      print\_kwargs(a=1)

{'a': 1} ⇒ 딕셔너리로 만들어져서 출력.

```
print_kwargs(name='foo', age=3)
```

즉, kwargs처럼 매개변수 이름 앞에 \*\*를 붙이면 매개변수 kwargs는 딕셔너리가 되고

모든 key=value의 형태같이 그 딕셔너리에 저장.

## 6) 함수의 결과값은 언제나 같다.

함수를 하나 보자.

```
def odd_and_mul(a, b):  
    return a+b, a*b
```

예기치 않은가? X

$a+b, a*b$ 은 같은 투포인트.

즉, result = odd\_and\_mul(3, 4)

result = (7, 12) 를 갖는다.

```
def odd_and_mul(a, b):  
    return a+b
```

예기치 않은가?

$a+b$ 은?

result = odd\_and\_mul(3, 4)

result = 7 이다.

return  $a+b$ 는 실행되지 않고 끝나버린다.

즉, 함수는 return을 만나는 순간 결과값을 반환하고 함수를 빠져나간다.

### [return 의 또 다른 쓰임새]

특별한 상황일 때 함수를 빠져나가고 싶으면 return을 명령으로 서서 즉시 함수를 빠져나갈 수 있다.

ex) def say\_nick(nick):

```
If nick == '바보':  
    return
```

print('%.s 는 나의 별명' %nick)

프로그램에서 예상

조건 만족시 빠져나가기

하는 가능성이 return을 사용.

## 7) 매개변수의 초기값 미리 설정

def say\_myself(name, man=True):

print('%.s 는 이름' %name)

함수들마다 매개변수

If man:

형상연결하는 것이 아닐 경우

print('남자입니다')

이렇게 함수의 초기값을 미리 설정하면 좋음.

else:

print('여자입니다')

Self-myself ('유성민') → 명시안하는 때는 초기값이 들어가고

Self-myself ('유성민', False)

이전식으로 평가하면 영시간 값이 들어간다.

\* 주의점 \*

def say\_myself(age, man=True, name)

↳ 이전식으로 중간에 위치하게 되면

명시안하는 경우 (man의 값을 바꿀 필요 없는 경우)

(25, 'yoo seong min') 이전식으로 코드를

쓸 때, 파일을 열어프리터는

'yoo seong min' 이런 변수를

man이 대입받지 name이 대입되는

알 수 없기 때문이다.

→ 오류발생.

초기화 시키고 싶은 변수는 항상 인자에 넣으자.

여러개면

Say-myself ('yoo seong min', 25, True, False)

이전식으로 자동처리되나요.

8) 함수에서 선언한 변수의 호석법(?)

지역변수 → 함수에서만 사용

$\alpha = 1$

def vartest( $\alpha$ )

$\alpha += 1$

$\alpha = 1$  이다. 뒤바뀜.

9) 함수내에서 함수 밖의 변수를 변경하는 방법.

두 가지가 있다.

1) return 사용.

a = 1

def vartest(a):

a += 1

return a

a = vartest(a)

2) global 명령어 사용.

a = 1

def vartest():

global a → 함수내부에서 함수 밖의 a변수를

a = a + 1 직접적으로 사용하게되는 곳.

vartest()

10) lambda

함수 생성하는 예약어

보통 한 줄로 간결하게 만들 때 사용

def를 사용하는데 짧은 경우에 불편하지 않거나

def를 사용할 수 없는 곳에 주로 사용

odd = lambda a,b : a+b

lambda는  
return a+b 결과값을 반환.

result = odd(3,4)

사용자 입력과 출력.

1) input의 사용.

사용자가 입력한 값을 어떤 변수에 대입하고 싶을 때.

a = input()

하고 입력값이면됨.

input()은 양쪽으로 모든 것을 문자열로 취급.

2) 프롬프트를 띠어서 사용자 입력 받기.

```
number = input("숫자를 입력하세요")
```

### 3) print 자세히 알기

① 큰 따옴표로 둘러싸인 문자열은 + 연산과 동일.

```
print ("Yoo" "Seong" "min")  
print ("Yoo"+ "Seong"+ "min")
```

] 같다! Yoo Seong min

② 문자열 피연산자는 문자로 조인다.

```
print ("Yoo", "Seong", "min")
```

Yoo Seong min

③ 한 줄의 결과값 출력.

```
매개변수 end로 지정  
예시 for i in range(10)  
print (~, end=' ')
```

```
print (i, end=' ')
```

## 파일 읽고 쓰기

### 파일 생성하기

```
f = open('새파일.txt', 'w')
```

파일 생성하기 위해 내장함수 open을 사용.

```
f.close()
```

open 함수는 입력으로 파일명과 열기모드를 받고

파일 객체 = open(파일명, 파일 열기 모드) 결과값으로 파일 객체를 돌려준다.

파일열기모드	기능
r	읽기모드 - 파일을 읽기만
w	쓰기모드 - 파일을 쓸 때
s	추가모드 - 파일의 마지막이나 새로운 내용 추가

파일을 쓰기모드로 열면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 삭제되고, 해당 파일이 존재하지 않으면 새로운 파일이 생성

경로를 지정하여 생성

C:/dok 디렉토리에 생성

f = open ('c:/doit / 서파일.txt', 'w')

f.close() 는 열려 있는 객체를 닫아주는 역할.

프로그램 종료시 파일이 열려 있는 파일의 객체를 자동으로 닫기 때문에

생략해도 되지만, close() 을 사용해서 직접 닫는게 좋다.

쓰기 모드로 열었던 파일을 단지 않고 다시 사용하려고 하면 오류가 발생.

파일을 쓰기 모드로 열어 출력 값 적기

f = open ("C:/doit / 서파일.txt", 'w')

for i in range (1, 11):

data = "x.d 번째 줄입니다. \n" + str(i)  $\Rightarrow$  틱이하게 쓰네.

f.write (data)  $\qquad \qquad \qquad$  이걸로 문자열

f.close()

$\gg> i = 10$

$\gg> string = "String contains x.d" + str(i)$

$\gg> String$

'String contains 10'

프로그램 외부에 저장된 파일을 읽는 여러가지 방법.

1) readline() 함수 이용

f = open ("C:/doit / test.txt", 'r')

line = f.readline()

print (line) 1번째 줄입니다.

f.close()

모든 줄을 읽어서 화면에 출력하기





```
args = sys.argv[1:]
```

```
for i in args:
```

```
    print(i)
```

sys 모듈의 argv는 명령창에서 입력 안드를 의미.

sys.py aaa bbb ccc

argv[0] argv[1] argv[2] argv[3]

## 간단한 스크립트

```
import sys
```

```
args = sys.argv[1:]
```

```
for i in args:
```

```
    print(i.upper(), end=' ')
```

소문자를 대문자로 바꾸어주는  
간단한 프로그램.

## 5) 파일 날개달기

클래스 사용장점: 같은 기능을 하는 변수와 함수를 묶은 코드집합이

반복하기 사용되는 경우, 유용하다.

클래스는 아니지만, 객체는 클래스로 만든 물품.

1개의 클래스는 무수한 객체를 만들고, 이들은 서로 통합된다.

클래스 구조 생성.

```
class <클래스명>:
```

각체에 놓자 지정할 수 있게 만들기.

class FourCal :

```
def setdata (self, first, second):  
    self. first = first  
    self. second = second
```

클래스 내에 구현한 함수는  
매개변수를 받는다.

a = FourCal ()

a. setdata (4, 2)

그럼에 정의와는 달리 매개변수를 오직만 받는다.

```
def setdata (self, first, second):  
    self. first = first  
    self. second = second
```

setdata 메서드의 첫 번째 매개변수 self에는

setdata 메서드를 호출한 객체 a가 자동으로 전달되기 때문.

클래스 내에 구현한 함수  
파이썬 '메서드'의 첫 번째 매개변수 이름은 관례적으로 self를 사용  
물론 self 말고 다른 이름을 사용해도 무방.

\* 메서드의 첫 번째 매개변수 self를 명시하는 것은  
다른언어와는 다른 파이썬만의 특별한 특징.

메서드의 또 다른 호출방법. (잘 사용하지는 X)

a = FourCal ()

FourCal. setdata (a, 4, 2 )

클래스 이름. 메서드 (객체, 인자1, 인자2 ... )

반드시 써넣기.

객체. 메서드 ( 인자1, 인자2, ... )

객체 반드시 생략.

def setdata (self, first, second):

    self. first = first

    self. second = second

a. setdata (4, 2)

    self. first = 4

    self. second = 2

!!

a. first = 4 → 수행시, a 객체의 객체변수 first가 4로

a. second = 2.

\* 객체에 생성되는 객체만의 변수를 객체변수라고 부름.

id ( a. first )

id는 객체의 주소를 반환하는 내장함수.

id ( b. first )

\* 객체변수는 다른 객체를 영향을 받지 않고 독립적으로 그 값을 유지.

숫자 지정 후 초기화가 가능 만들기.

class FourCal :

    def setdata (self, first, second):

self. first = first

self. second = second

def add (self) :

result = self.first + self.second

return result

생성자 (Constructor)

위에서 봤던 FourCal 클래스의 핵심은

seedata를 수행한 후에 add를 해야 유효가 됨을

겁니다.

파이썬 예제의 이름을 \_\_init\_\_ 을 사용하면

이 예제는 생성자가 된다.

생성자가 클래스에 존재하면,

생성자에서 필요한 매개변수를 반드시

직접 생성 시 넘겨 주야 한다. 아니면 어려.

def \_\_init\_\_(self, first, second):

self. first = first

self. second = second

직접 생성 시, a = FourCal () → 에러

a = FourCal (4, 2)

## 클래스의 상속.

어떤 클래스를 만들 때, 다른 클래스의 기능을 물려받을 수 있게

class MoreFour Col (Four Col) :

클래스명 상속할 클래스 이름.

이제 MoreFour Col 클래스는 Four Col 클래스를 상속 받았으므로,  
Four Col의 모든 기능 사용할 수 있다.

상속이 필요한 이유

: 보통 상속은 기존 클래스를 변경하지 않고 기능을 추가하거나  
기존 기능을 변경하려고 할 때 사용.

클래스에 기능 추가하는데 있어 굳이 상속?

기존 클래스가 이미 보유한 흥미로 제공되거나, 수시로 사용되는  
많은 상황이라면 상속해야 한다.

메서드 오버라이딩

: 부모 클래스에 있는 메서드를 동일한 이름으로 다시 만드는 것을  
overriding (덮어쓰기)라고 하면 오버라이딩 하면,  
부모 클래스의 메서드 대신 오버라이딩한 메서드가 호출된다.

클래스 변수 : 클래스 안에 변수를 선언하여 생성.

( 클래스 이름 . 클래스 변수 의 형식으로 사용.  
개체 . 클래스 변수 )

클래스 변수의 특징으로는 클래스로 만든 모든 객체가 공유됨.

수집하면 내용이 바뀜.

## 모듈

함수나 변수 또는 클래스를 모아놓은 파일을 일컫는다.

모듈은 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만든 파일이다.

파이썬 확장자 .py로 만든 파일은 모두 모듈이다.

## 모듈 불러오기

Import 명령어는 이미 만들어져 있는 파이썬 모듈을  
사용할 수 있게 해주는 기능을 한다.

Import mod1

mod1.add(3, 4) → 이렇게 모듈.함수 이름으로 사용

→ 설치 시, 자동으로 깔리는  
파이썬 라이브러리

Import는 현재 디렉토리에 있는 파일이나 파이썬 라이브러리  
가 저장된 디렉토리에 있는 모듈만 불러올 수 있다.

from 모듈이름 import 모듈함수

from mod1 import add

add(3, 4)

위와 같은 형식으로 모듈이름 생략 가능

from mod1 import add, sub

from mod1 import     
또는 함수.

If \_\_name\_\_ == "\_\_main\_\_": 의 의미.

mod1.py

```
def add(a,b)  
    return a+b  
print(add(3,4))
```

그리고 import mod1을 하면

7이 출력된다.

mod1.py가 실행되기 때문.

이런 문제를 방지하기 위해서.

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

```
if __name__ == "__main__":  
    print(add(3,4))
```

이 문장을 추가.

\_\_name\_\_ 변수란?

파이썬의 \_\_name\_\_ 변수는 파이썬이 내부적으로 사용하는  
특별한 변수의 이름이다.

Python mod1.py 처럼 직접 mod1.py를 실행하는 경우

mod1.py의 \_\_name\_\_ 변수에는 \_\_main\_\_ 이 저장

하지만 Python shell이나 다른 python module에서  
모듈을 import 하는 경우 mod1.py의 \_\_name\_\_에는  
mod1이 저장.

클래스나 변수 등을 포함한 모듈.

1) 클래스 사용

a = mod1. math()  
~~~~~ 모듈. 클래스()

2) 변수

mod1.PI

다른 파일에서 모듈 불러오기

self 다른 디렉토리에 있을 때. 모듈 불러오기

1. sys.path.append (모듈 저장한 디렉토리)

import sys

sys.path는 파일 실행 디렉토리가 위치되어 있는 디렉토리를 보여준다.

여기서 모듈을 저장한 디렉토리의 경로를 추가시킨다.

sys.path.append ("C:/date/mymod")  
리스트.

2. PYTHONPATH 환경 변수 사용

set PYTHONPATH = C:/date/mymod

와! 명령어를 터미널에서 사용해서  
별도의 모듈 추가 작업 없이 원하는 모듈을 불러와서  
쓸 수 있다.

파이썬은 무엇인가?

.을 사용해서 파이썬 모듈을 계층화(파악하기 편)로 관리할 수 있다.

모듈들이 A, B 인 경우

A는 패키지 이름

B는 A의 하위의 B 모듈.

파이썬 패키지는 디렉터리와 파이썬 모듈로 이루어 진다.

가상의 game 패키지 예

```
game/  
  __init__.py  
sound/  
  __init__.py  
  echo.py  
  wav.py  
graphic/  
  __init__.py  
  screen.py  
  render.py  
play/  
  __init__.py  
  run.py  
  test.py
```

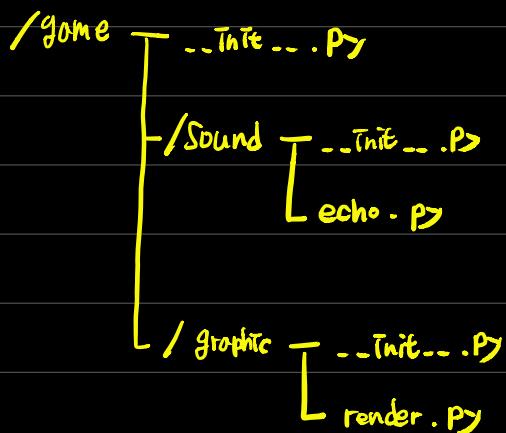
game, sound, graphic, play는 디렉터리 이름이고 확장자가 .py인 파일은 파이썬 모듈이다. game 디렉터리가 이 패키지의 루트 디렉터리이고 sound, graphic, play는 서브 디렉터리이다.

간단한 파이썬 프로그램이 아니라면, 이렇게 패키지 구조로  
파이썬 프로그램을 만드는 것이 공동작업이나 유지 보수 등  
여러 면에서 유리

또는 패키지 구조 모듈을 만든다면 다른 모듈과  
이름이 겹쳐도 양립하기 사용 가능.

파키지 안의 함수 실행.

3 가지가 있다.



Set PYTHONPATH = C:/doit

echo.py 파일의 echo-test 함수 실행.

1. echo 모듈을 Import 하여 실행하는법.

```
>>> import game.Sound.echo  
>>> game.Sound.echo.echo_test()
```

2. echo 모듈이 있는 디렉토리 까지 from ... Import

```
>>> from game.Sound import echo  
>>> echo.echo_test()
```

3. echo-test()를 직접 Import

```
>>> from game.Sound.echo import echo_test  
>>> echo_test()
```

\_\_init\_\_.py → \_\_init\_\_.py만 참조 가능.

```
[ import game  
    game.sound.echo.echo-test()
```

이 코드는 예외

- Import game.sound.echo.echo-test()

이 코드로 예외

. 를 이용해서 Import a.b.C 처럼

Import 할 때 가장 마지막 항목인 C는 반드시

모듈이나 패키지여야 한다.

\_\_init\_\_.py 의 용도.

: 해당 디렉토리가 패키지의 일부임을 알려주는 역할

만약 game, sound, graphic 등 패키지가

포함된 디렉토리가 \_\_init\_\_.py 파일이 없으면 제대로 인식되지 X.

- Python 3.3 이전 버전은 \_\_init\_\_.py 없어도 패키지로 인식

그러나 하위 디렉토리 포함은 \_\_init\_\_.py 파일 써놓는 것에 안전.

```
from game.Sound import *
```

echo.echo-test() ⇒ Name Error

이렇게 특정 디렉토리의 모듈을 \*를 사용해서 Import 할 때는

다음과 같이 해당 디렉토리 \_\_init\_\_.py 파일에

\_\_all\_\_ 변수를 설정하고 Import 할 수 있는 모듈을

정의해 주어야 한다.

\_\_init\_\_.py 파일

\_\_all\_\_ = ['echo']

여기서 ...all... 이쓰이기에는 굳이 sound 디렉토리에서

\* 기호를 사용해서 import 할 경우 이곳에 정의된

echo 모듈만 import 된다는 의미.

### \* 각각하기 쉬운데

from game.sound.echo import \* 는 ...all... 과 상관없이

무조건 import 된다. 이렇게 ...all... 과 상관없이!

무조건 import 되는 경우는 from a.b.c import \*에서

from의 마지막 항목인 C가 모듈인 경우이다.

### relative 패키지

단독 graphic 디렉토리의 render.py 모듈이 sound 디렉토리의

echo.py 모듈을 사용하고 싶다면 어떻게 할까?

다음과 같이 render.py를 수정하면 가능.

# render.py

from game.Sound.echo import echo-test

### relative하기 import 하기

from ..sound.echo import echo-test

def render-test():

.. 는 부모 디렉토리를 의미하고,

여기서 graphic과 sound는 동일한 depth이거나 그므로

.. 부모디렉토리를 사용하여 import가 가능.

[ .. 부모  
.. 혼자

\* 인터프리터에서 `below` 접근자 사용시 아래.  
모든 아래/한 사용한 것.

## 예외처리

오류 예외 처리 기법.

### 1. try, except 문

try :

...

except [ 발생오류 [as 오류 메시지 변수] ] :

... [ ]는 생략 가능함 의미

try 수행 중 오류발생 시 except가 수행.

### 1. try :

...

except:

...

} 이 경우는 오는 오류의 종류에 상관없이  
except 블록을 수행.

### 2. 발생오류만 포함

try :

...

except 발생 오류 :

...

### 3. 발생오류와 오류 메시지 변수까지 포함.

try :

except 발생 오류 as 오류 메시지 변수 :

...

(예시)

try:

4 / 0

except ZeroDivisionError as e:

print(e)

try .. finally

try 문에는 finally 절을 사용할 수 있다.

finally 절은 try 를 수행 도중 예외 발생 여부에 상관없이 항상 수행된다.

보통 finally 절은 사용한 리소스를 close 할 때 많이 사용

f = open('foo.txt', 'r')

try:

...

finally:

f.close()

여러개의 오류 처리하기

try 문 안에서 여러 개의 오류를 처리하기 위해 다음 구문을 사용.

try:

...

try:

except  $E^21$ :

...

1.  $E^21$  처리

...  
except ( $E^21, E^22$ ) as e:

except  $E^22$ :

...

## 오류 처리하기

except  $E^21$  pass 를 써서 무시한다.

## 오류 일으켜 발생시키기

: raise 명령어

이를 하면 꼭 작동하는 예외를 발생시킬 수 있게 오류를 만들 때.

class Bird:

def fly(self):

raise NotImplementedError

↳ 이렇게 사용 오류로

꼭 작동하는 부분구현  
안드로이드를 때 발생하는 예외.

class Eagle:

pass

e = Eagle()

e.fly() → 예외

구현을 안했지.

## 예외 만들기

: 프로그램 수행 도중 특수한 경우에만 예외 처리를 하기 위해서  
종종 예외를 만들어서 사용, Exception 클래스를 상속해서 가능.

```
class MyError (Exception):  
    pass
```

```
def say_nick(nick):  
    if nick == '바보':  
        raise MyError()  
    print(nick)
```

예외처리로 오류 메시지까지 출력된다.

```
try:  
    say_nick('친구')  
    say_nick('바보')  
except MyError as e:  
    print(e)
```

처리할 수 없는 예외 발생.

오류 메시지를 보여주기 위해서,  
오류 클래스가 \_\_str\_\_ 메소드 구현해야 함.

```
class MyError (Exception):  
    def __str__(self):  
        return '처음하지 않는 별명'  
        # 전식으로 말하자.^
```

- Python 스텝1 끝 -  
2020. 3 - 28 토.  
AM 61:20  
김현수. 씨발 쿠

이후로는 배장함수, 외장함수 자주 쓰는 것을 여기에 메모하도록 ..