파이썬 기초

1. 프로그래밍에 대한 이해

파이썬은 프로그래밍 언어, 그렇다면 프로그래밍이라는 것은 무엇인가?

프로그래밍은 프로그램을 만드는 것! 그렇다면 프로그램은 무엇인가?

pro / gram : pro = forward, before, gram = record → 미리 작성된 프로그램

(컴퓨터 프로그램의 예) 카카오톡에서 상대방에게 메세지 전송 버튼을 누르면 서버에 메세지가 전송되어 상대방에세 전달되는 과정

컴퓨터가 이해하는 언어 ≠ 사람이 이해하는 언어(기본 체계가 다르다

컴퓨터가 이해하는 언어는 '이진 숫자(0과 1로 이루어짐)'로 이루어져 있다.

▸ 이진 코드: 이진 숫자로 이루어진 코드

▸ 프로그래밍 언어: 사람이 이해하기 쉬운 언어로 프로그램을 만들기 위해 만들어짐

(why? 사람은 컴퓨터의 언어를 이해하기 어렵기 때문)

▸ 소스 코드: 프로그래밍 언어를 작성한 프로그램(사람이 이해하기 쉽게)

▸ 코드 실행기: 사람이 작성한 프로그래밍 언어를 컴퓨터가 이해할 수 있도록

이해하기 쉽게 프랑스인과 한국인이 대화하기 위해 필요한 '번역기'

이 번역기가 바로 코드 실행기라고 보면 된다.

2. 파이썬 개발환경 만들기

▸ 텍스트 에디터: 프로그래밍 언어로 이루어진 코드를 작성 (예: Colab, IDLE)

▸ 코드 실행기: 텍스트 에디터가 작성한 코드를 실행

▸ 통합개발환경(IDE): 텍스트 에디터 + 코드 실행기

※ 프로그래밍 언어에서는 반드시 텍스트 에디터와 코드 실행기가 필요

▸ interactive shell(대화형): 컴퓨터와 대화하듯이 코드를 한 줄씩 입력 실행하여 결과를 볼 수 있다. (예: Python)

3. 파이썬에서 사용하는 용어

▸ 표현식 → 문장 → 프로그램

표현식: 값으로 표현하는 코드(단순 숫자나 문자(열))이며, 실행이 될 수는 없다.

문장: 실행될 수 있는 코드(파이썬에서는 실행되는 모든 한 줄 코드가 하나의 문장)

프로그램: 문장들이 모인 것

▸ 식별자를 잘 구별하기 위한 작성 방법 : why? 가독성을 높이기 위해

a. 스네이크케이스: 모두 소문자이거나 대문자인 평평한 모양
   스네이크케이스의 경우에는 space bar 공간을 under bar(_)로 작성
   (예) class_name, HOUSE_TYPE
b. 캐멀케이스: 단어의 첫 글자는 대문자이고 나머지는 소문자인 울퉁불퉁 모양
   space bar 공간을 대문자로 작성
   (예) MyName, HouseType

4. 파이썬 자료형(=data type)

python의 '기본' 자료형은 크게 세 가지로 나누어진다: 문자열, 숫자, 불(논리연산자)

▸ python에서 실제로 사용하는 약어

a. 문자열= string

b. 숫자 中 정수 = int (integer)

c. 숫자 中 실수 = float

d. 논리연산자 = bool(boolean) : True OR False

자료형 구분이 중요한 이유? 데이터 처리 과정에서 중요하므로

(예) 평균을 구하기 위해서는 숫자 자료형이 적합하다.

올바른 데이터 분석을 위해서는 데이터 자료에 맞는 데이터 처리가 이루어져야 한다.

5. 사칙 연산자

※ 사칙연산자는 데이터 타입에 따라 서로 다른 역할 (단, 연산자 우선순위는 동일)

사칙연산에서 문자와 숫자는 '함께', '동시에' 쓸 수 없다.

(유의) print("안녕" + 1) ≠ print("안녕" + "1")

전자의 경우 1은 숫자(int) 데이터 타입이고, 후자의 1은 문자(string) 데이터 타입

a. 숫자 데이터 타입: 기존의 수리 사칙연산으로 적용된다.

b. 문자 데이터 타입: 텍스트 연결**(+) OR** 반복**(*)**으로 적용된다

문자 데이터 타입의 경우에는 +와 *만 사용할 수 있다.

연산자 '+' = 데이터의 연결

연산자 '*' = 데이터의 반복 (순서가 어디에 있던 같은 값을 출력한다)

(예) print("안녕" * 5) = print(5 * "안녕")

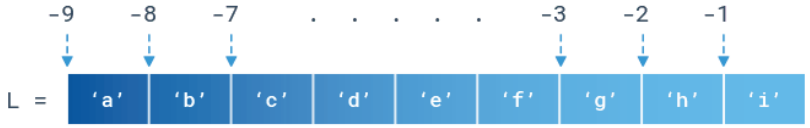
6. 인덱스

인덱스: 자료의 위치 (값 = 숫자일 수도 있고, 문자일 수도 있다.)

파이썬에서 인덱스를 사용하기 위해 대괄호[ ]를 사용

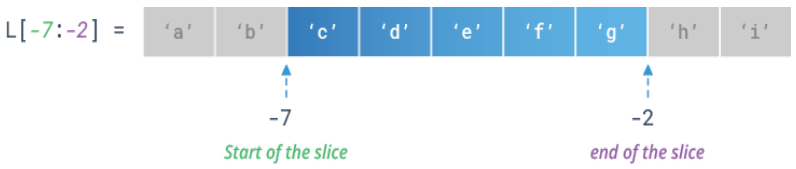
★ 파이썬은 제로 인덱스

인덱싱: 인덱스를 이용하여 자료를 추출하는 것



▸ 슬라이싱: 인덱스의 범위를 이용하여 인덱싱하는 것

시작과 끝은 콜론(:)으로 연결한다. ※단, 파이썬은 끝 번호를 포함하지 않음



print(a[3:-3])

앞에서부터 슬라이싱하는 것과 뒤에서부터 슬라이싱하는 것 혼용해서 사용 가능 (중간 글자를 출력할 때 유용)

(예) a = "PythonLanguage"

print(a[7:-3]) → angu

7. 함수 len( )

소괄호 안에 문자열 데이터를 넣으면 해당 문자열 길이 값을 반환

문자열 길이 값 = 문자열의 문자 개수

★ 여러 개의 함수가 있는 경우, 안 → 밖의 순서로 실행된다.

▸ 함수 input( )

프롬프트에서 사용자가 입력한 값을 받을 때 사용

(예) input("값을 입력하세요> ")

함수 input( )이 실행될 때는 사용자가 값을 입력하기 전까지 프로그램이 잠시 멈추는데 이것을 블록이라고 함 →

이 상태는 값을 입력하는 부분에 있는 커서로 확인 가능

※ **input**을 통해 입력한 값은 무조건 문자열로 저장 **(**숫자로 입력하더라도 문자열로 인식**)**

8. 변수

변수는 '변하는 값' ≠ 상수

(예) a = 1 → print(a) ; 출력값 1

a = 1 / a = 2 → print(a) ; 출력값 2

∴ a 식별자는 변하는 값 **'변수'**

변수는 원하는 모든 자료형의 값을 저장할 수 있다.

이때 변수의 이름은 직관적인**(=**누구나 보아도 이해할 수 있는**)** 이름으로 지정하는 것이 좋다.

변수 이름에 값을 저장하기 위해 할당 연산자**(=)** 기호를 사용한다.

9. 문자열, 숫자 관련 함수

▸ 변수 vs. 함수? 괄호의 유무를 살피면 된다.

파이썬에서 변수 뒤에는 소괄호**( )**가 없지만**,** 함수 뒤에는 소괄호가 있다**.**

(예) a = 10 ; 이때 a는 변수

input( ) ; input은 함수

▸ 함수의 구성: input, function, output

★ 프로그램을 설계할 때 return값인 output이 반드시 나오도록 해야 한다.

▸ 함수 format( ): 값을 순차적으로 할당해준다.

사용방법: 중괄호**{ }**를 큰따옴표 **" "**로 묶어준 다음 마침표로 연결시켜 사용 가능

(예) str = "{}".format(23); print(str); print(type(str))

실행결과: 23 <class 'str'>

※ str을 변수로 사용할 경우, 오류가 발생할 수 있다. (why? str은문자열을 뜻하는 키워드)

이럴 땐 런타임 다시시작을 눌러서 실행

※ format 함수 사용 시 유의점: 중괄호 안에 있는 값 ← **format** 함수 안에 있는 값

▸ 정수형 숫자를 문자열로 반환

기본 포맷 → **“{:d}”.format( )**

(a) 숫자만큼 자릿 수를 반환하기 → “{:숫자d}”.format( ) (예) “{:5d}”.format(23) → 결과값 _ _ _ 23

(b) 자릿수와 남은 자리 0으로 채우기 → “{:0숫자d}”.format( ) (예) “{:05d}”.format(23) → 00023

(c) -는 자릿수 1을 차지한다 → (예) “{:05d}”.format(-23) → -0023

(d) 양수 기호를 출력하기(=별도 지정이 없을 경우 양수는 출력 안됨)

“{:+d}”.format( ) (예) “{:+d}”.format(23) → +23

▸ 자릿수와 부호를 함께 지정하는 경우 ※ 부호도 자릿수 **1**을 차지

(i) 자릿수 & 숫자 바로 앞에 기호 생성 & 남은 자리 공백

"{:**+**숫자d}".format( ) (예) "{:+5d}".format(23) → _ _ +23

(ii) 자릿수 & 맨 앞에 기호 생성 & 남은 자리 공백

“{:**=**+5d}”.format( ) (예) “{:=+5d}”.format(23) → + _ _ 23

(iii) 자릿수 & 맨 앞에 기호 생성 & 남은 자리 0

“{:**=**+05d}”.format( ) (예) “{:=+05d}”.format(23) → +0023

(e) 음수 기호 출력하기 (=별도 지정이 없는 경우에도 음수는 출력됨)

“{:+d}".format( ) (예) “{:+d}”.format(-23) → -23

"{:-d}".format( ) (예) "{:-d}”.format(-23) → -23 ※ **±** 어떤 기호를 집어넣던 음수는 출력 **必**

“{:d}".format( ) (예) “{:d}”.format(-23) → -23

▸ 실수형 숫자를 문자열로 반환 → 실수형은 정수형으로 변환 불가 **(**변환하고 싶을 때 캐스트 함수를 **先)**

기본 포맷 → **“{:f}”.format( )**

(a) “{:f}”.format( ) (예) “{:f}”.format(23.45) → 23.450000 (※ float default value = 6)

(b) 자릿수 & 남은 자리는 빈칸

“{:15f}”.format( ) (예) “{:15f}”.format(23.45) → 23.450000 ※ 소수점도 **1**자리 차지

(c) 자릿수 & ±기호 & 남은 자리를 빈칸으로

“{:+15f}”.format(23.45) OR “{:-15f}”.format(23.45) → 23.450000

(d) 자릿수 & ±기호 & 남은 자리를 0으로 채움

“{:+015f}”.format(23.45) OR “{:-015f}”.format(23.45) → +0000023.450000

(e) 소수점 아래 자릿수 지정하기 → 보이고 싶은 자릿수를 기준으로 작성

"{:15.**3**f}".format(23.456) → 23.456

"{:15.**2**f}".format(23.456) → 23.46 (소수점 셋째자리에서 반올림→ 표기는 둘째자리까지)

"{:15.**5**f}".format(23.456) → 23.45600 ※ 소수점 자리가 줄어든만큼 0으로 대체

※ 파이썬의 반올림 특징

(1) 반올림 기준은 5가 아니라 6이다

(예) "{:15.1f}".format(23.45) → 23.4

"{:15.1f}".format(23.46) → 23.5

(2) 양 끝 값을 기준으로 정확히 딱 절반의 위치에 있을 경우 가장 가까운 짝수의 값을 반환

(예) "{:15.0f}".format(2.5) → 2

"{:15.0f}".format(3.5) → 4

▸ 의미없는 소수점 제거하기

기본 포맷 → **“{:g}”.format( )**

1. 소수점 이하 표기 방식 : 디폴트값은 소수점 이하 여섯자리

2. 별도의 표기를 하지 않는다? 소수점 여섯자리까지 표기

3. 3.56의 경우 → 3.560000

4. 이때 g를 사용해주면 3.56 이렇게만 출력

▸ format 함수와 f문자열

“{ }”.format( ) vs. f”{ }

print("{}".format(23)) → 23 = print(f"{23}") → 23

print("{}".format(2 + 3)) → 5 = print(f”{2+3}”) → 5

※ 어떤 때 format or f문자열을 사용하는 것이 좋을까?

(1) 문자열이 많을 때

```
mc = "유재석"
reward = 100
"""\
유퀴즈 프로그램은 다양한 사람들을 만나서 이야기하는 토크쇼이다.
MC는 {}이고, 퀴즈를 맞추면 상금 {}만원을 준다.\
""".format(mc,reward)
mc = "유재석"
reward = 100
f"""\
유퀴즈 프로그램은 다양한 사람들을 만나서 이야기하는 토크쇼이다.
MC는 {mc}이고, 퀴즈를 맞추면 상금 {reward}만원을 준다.\
"""
# 출력값은 동일하지만 문자열이 많은 경우 format 함수를 사용하는 것이 더 간결하고 편하므로
문자열이 많은 경우에는 f문자열이 아닌 format 함수를 사용한다.
```

(2) 여러 값을 리스트 형태로 사용할 경우

```
# data = ["강백호", 17, "M", "난 천재니까"]
# """\
# 이름: {},
# 나이: {},
# 성별: {},
# 좌우명: {}\
# """.format(data[0],data[1],data[2],data[3])
# format( )을 사용할 경우 전개연산자 데이터는 *을 활용하면 된다.
data = ["강백호", 17, "M", "난 천재니까"]
```

```
"""\
이름: {},
나이: {},
성별: {},
좌우명: {}\
""".format(*data)
# compare
data = ["강백호", 17, "M", "난 천재니까"]
f"""\
이름: {data[0]},
나이: {data[1]},
성별: {data[2]},
좌우명: {data[3]}\
""" ;
```

▸ 대/소문자 변환: upper( ), lower( )

전체 대문자로 변환하기 : upper( )

string = “HEllo” → print(string.upper()) : HELLO

전체 소문자로 변환하기 : lower( )

print(string.lower()) : hello

※ 앞 글자는 대문자, 뒤부터는 소문자로 출력하기 **: swapcase( )**

print(string.swapcase()) : Hello

▸ 공백 제거

string = “ 안녕 “

(1) 양쪽 공백 모두 제거 : strip( )

print(string(strip()) → 안녕

(2) 왼쪽 공백 제거 : lstrip( )

print(string(lstrip()) → 안녕_____ (뒤에는 공백 있음)

(3) 오른쪽 공백 제거 : rstrip( )

print(string(rstrip()) → 안녕

▸ 문자열 구성 확인 : **.is ~ ()**

s.isalnum() # 문자열이 알파벳 또는 숫자로 구성되었는지 확인 ※ 한글, 알파벳, 숫자 *True*, 특수문자 *False*

s.isdigit() # 문자열이 숫자로 인식될 수 있는지

s.isupper() # 문자열이 대문자로 구성되어 있는지 확인

s.islower() # 문자열이 소문자로 구성되어 있는지 확인
s.isalpha() # 문자열이 알파벳으로 구성되어 있는지 확인 ※ 한글, 알파벳 *True*, 특수문자 *False*
s.isdecimal() # 문자열이 정수형으로 구성되어 있는지 확인
s.isidentifier() #문자열이 식별자로 사용될 수 있는지 확인
s.isspace() # 문자열이 공백으로 되어 있는지 확인

▸ 문자열 첫 번째 위치 반환 : find( ) / rfind( )

기본적으로 find 함수는 좌 → 우 방향으로 이루어진다.
a = "hello python"
print(a.find("l")) → result : 2

rfind 함수는 우 → 좌의 방향으로 이루어진다.
print(a.rfind("l")) → result : 3

▸ 원하는 데이터를 찾고 싶을 때 : in 연산자

기본형식 : 찾고 싶은 문자 in 찾으려는 문자열
print("안녕" in "안녕하세요") → result : True
print("잘자" in "안녕하세요") → result : False

→ 딕셔너리의 키 존재 여부를 알려주는 프로그램
```
key = input("찾고 싶은 키를 입력하세요: ")
if key in bs:
  print("존재하는 키 입니다. 해당 값은 ", bs[key])
else:
  print("존재하지 않는 키 입니다.")
result:key
```

▸ 문자열을 특정 문자로 잘라서 리스트로 반환하기 : **split**

※ **split( )** 함수는 문자열 데이터 타입을 무조건 **"리스트"**로 반환한다는 거 잊지 않기**!**
s = "1 2 3 4 5"
print(s.split(" ")) → result : ['1', '2', '3', '4', '5']
print(p.split("-")) → result : ['010', '1234', '5678']

10. 조건문

▸ 조건문은 bool 자료형 = **True / Flase** 값만 가질 수 있다. = 비교연산자 사용

a. 같다 ==
b. 같지 않다, 다르다 !=
c. 작다, 미만 <
d. 크다, 초과 >

e. 작거나 같다, 이하 <=

f. 크거나 같다, 이상 >=

▸ 비교연산자

a. 문자열에 적용할 경우 글자 순서가 앞에 있는 값이 작은 값

(예) print("가" > "나") → False

b. 범위 값으로 비교

(예) x = 25; print(10 < x < 30) → True

▸ 논리연산자

A	B	A AND B	A OR B	NOT A
A	B	A && B	A || B	!A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

▸ 조건문 사용하기

조건문 : if ~ (elif) ~ (else) ~ → 조건을 만든 후, 조건의 참 거짓 여부에 따라 실행 코드가 달라지는 구조

(1) 기본

print(True) → 결과값: True / print(type(True)) → 결과값: <class 'bool'>

print(False) → 결과값: False / print(type(False)) → 결과값: <class 'bool'>

print(1 == 2) → 결과값: False / print(1 != 2) → 결과값: True

print(1 < 2) → 결과값: True / print(1 >= 2) → 결과값: False

print("가" == "나") → 결과값: False / print("가" != "나") → 결과값: True

print("가" > "나") → 결과값: False / print("가" < "나") → 결과값: True

(2) not 사용하기

print(True) <-> print(not True) # = False와 같은 의미

```
x = 5
print(x < 10)
print(not x < 10) # ; 조건의 결과(True)와 반대의 값을 출력 (x < 10의 조건이 먼저 수행된 다음 not)
```

(3) 단일 조건문 : 참일 때만 수행되며 거짓일 땐 pass 된다.

▸ True 조건 하나만 있을 경우

```
number = int(input("정수를 입력하세요: "))
if number > 0:
  print("입력한 정수값은 양수입니다.")
```

▸ True 조건을 2개 이상 설정할 경우

```
number = int(input("정수를 입력하세요: "))
```

```
if number > 0:
  print("입력한 정수값은 양수입니다.")
if number == 0:
  print("입력한 정수값은 0입니다.")
```

▸ True 조건이 1개 (이상) & 거짓의 실행문 O → if ~ (elif) ~ else 조건문

(예) 학점을 알려주는 프로그램) 구간: A: 3.5 ~ 4.5 / B: 2.5 ~ 3.5 / C: 1.5 ~ 2.5 / D: 0.0 ~ 1.5 / F: 0.0

```
score = float(input("학점을 입력하세요: "))
if 4.5 >= score > 3.5:
  print("A학점입니다.")
elif 3.5 >= score > 2.5:
  print("B학점입니다.")
elif 2.5 >= score > 1.5:
  print("C학점입니다.")
elif 1.5 >= score > 0.0:
  print("D학점입니다.")
else:
  print("F학점입니다.")

# 코드를 더욱 간결하게 작성하는 방법
# why? 비교의 횟수를 줄여 가동의 효율성을 높이기 위함
score = float(input("학점을 입력하세요: "))
if 4.5 >= score > 3.5:
  print("A학점입니다.")
elif score > 2.5:
  print("B학점입니다.")
elif score > 1.5:
  print("C학점입니다.")
elif score > 0.0:
  print("D학점입니다.")
else:
  print("F학점입니다.")
# 위에서 검사가 된 것은 생략할 수 있다 → 8회의 비교를 5회의 비교로 축소
# 결론: 조건 3개 이상 연결할 때, 값의 전체 범위가 아닌 하위 값만 비교하면 더 효율적 코딩이 가능하다.
(예) 3.6 ~ 4.5의 경우 A 학점으로 분류해놓았으므로, 2.5 ~ 3.5가 아닌 2.5까지의 범위만 지정해도 가능하다.
```

▸ 조건문에 들어가면 False로 바뀌는 값 : None, 정수 0, [] → 빈 리스트

이 경우를 제외한 값은 True로 출력된다.

▸ **Pythonic**한 코드로 작성하기 → 최대한 간결하게

```
# [조건이 참인 경우의 값] if [조건] else [조건이 거짓인 경우의 값]
def solution (num1, num2):
    return 1 if num1 == num2 else -1
```

▸ pass 키워드 : 개발 중 나중에 구현하고자 할 때 사용 (단, 에러 없이 지나가므로 사후 확인이 어려움)

```
no = input("숫자(정수)를 입력하세요: ")
last_no = int(no[-1])
if last_no % 2 == 0:
else:
```

<출력값> - *ERROR* 발생*!!!*

IndentationError: expected an indented block after 'if' statement on line 4

★ 이때, pass 키워드를 사용하면 에러 없이 결과물을 출력할 수 있다.

```
no = input("숫자(정수)를 입력하세요: ")
last_no = int(no[-1])
if last_no % 2 == 0:
  pass
else:
  pass
```

pass 키워드는 보통 **'**사후에**'** 다시 코드를 작성하고자 남겨둘 때 사용한다**.** 별도의 에러문구 없이 코딩을 이어갈 수 있다는 장점이 있지만, 그와 동시에 별도의 '에러'가 출력되지 않는다는 점에서 개발의 과정에서 종종 놓칠 수 있다는 문제가 발생한다. 이러한 문제를 해결하기 위해 개발자가 '일부러' 에러를 발생시킨다.

▸ 확인을 위한 에러가 필요**! : raise NotImplementedError**

```
no = input("숫자(정수)를 입력하세요: ")
last_no = int(no[-1])
if last_no % 2 == 0:
  raise NotImplementedError
else:
  raise NotImplementedError
```

<실행 결과>

숫자(정수)를 입력하세요: 265

NotImplementedError Traceback (most recent call last) 이와 같은 에러 출력

11. 리스트: 개별값이 쉼표로 구분되어 대괄호 안에 있는 것

a. 여러 가지 데이터를 다룰 때 하나의 변수에 많은 값을 집어넣는 경우
b. 자료들을 모아서 사용할 수 있음(서로 다른 데이터 타입의 값도 하나의 리스트 안에 넣을 수 있다)
c. 대괄호 내부에 자료를 넣어 선언

(1) 리스트 생성 : 리스트 안의 값, 즉 대괄호 안의 개별값을 '요소'라고 부른다

```
list_1 = [1, 2, 3, 4, 5]                    ; print(list_1) →  실행결과: [1, 2, 3, 4, 5]
list_2 = ["안", "녕", "하", "세", "요"]      ; print(list_2)   실행결과: ['안', '녕', '하', '세', '요']
```

(2) 인덱스를 활용한 리스트 내부 요소 사용

→ 리스트 vs. 튜플? 리스트는 내부 요소 변경이 가능하지만, 튜플은 변경이 불가능

list_a = [[1, 2, 3, 4], "안녕하세요", False] # 해당 리스트의 요소는 총 3개

print(list_a) → [[1, 2, 3, 4], '안녕하세요', False] / list_a[0] → [1, 2, 3, 4]

리스트 안에 리스트 접근 접근하는 방법

print(list_a[0][2]) 출력값: 3 / print(list_a[1][2]) 출력값: 하

(3) 슬라이싱 : 파이썬은 제로인덱싱 & 끝 번호는 포함하지 않는다.

print(list_a[0:2]) [[1, 2, 3, 4], '안녕하세요'] → 인덱스 0 & 인덱스 1의 값이 출력 (why? 2는 포함 X)

▸ 시작번호 생략 = 자료를 처음부터 마지막 지정까지 출력

print(list_a[:2]) [[1, 2, 3, 4], '안녕하세요']

▸ 끝번호 생략 = 자료를 처음 지정부터 끝까지 출력

print(list_a[1:]) ['안녕하세요', False]

▸ 음수 인덱싱 : 맨 끝자리가 -1로 시작된다. print(list_a[-1]) False

(4) 리스트 요소 변경

list_a[2] = True # 기존 False에서 True로 값을 변경

list_a[1] = "Hello"

list_a[0][0] = 10

▸ 리스트 연산자: 요소 추가 ★비파괴적 처리(사칙연산) ***vs.*** 파괴적 처리(함수)

※ 비파괴적 처리와 파괴적 처리가 무엇이지?

a. 비파괴적 처리 : 원본 데이터나 객체를 변경하지 않고, 새로운 데이터 또는 객체를 생성하는 작업
즉, 예를 들어 list1 + list2 를 하였다고 이 값 자체가 영구저장 되는 것이 아니다. (별도의 변수를 지정했을 경우 해당 값으로 할당되어 영구 저장)

→ 연결(+), 반복(*), 길이(len)

(1) 연결을 위한 '+' 사용

list_1 = [1, 2, 3] / list_2 = [4, 5, 6]

print(list_1 + list_2) 결과값: [1, 2, 3, 4, 5, 6]

(2) 반복을 위한 '*' 사용

print(list_1 *3) result: [1, 2, 3, 1, 2, 3, 1, 2, 3]

결론 : +와 * 는 변수의 본래값에 영향을 미치지 않는 '비파괴적 처리'이다.

len(list_1) # 요소의 개수 result: 3 (처음 할당된 값과 동일)

b. 파괴적 처리 : 원본 데이터나 객체를 '직접 수정' 또는 '변경'하는 작업을 의미한다. 이러한 작업은 데이터나 객체 자체를 변경하여 결과적으로 이전의 상태를 변경시킨다.

인덱스 마지막 값 이후에 추가 = append (즉, ***append***는 별도의 인덱스 지정이 필요없음)

list_1.append(5) result: [1, 2, 3, 5]

중간에 값을 추가 = 인덱스 번호를 통해 원하는 위치에 추가

list_1.insert(3, 4) # 인덱스 3번 위치에 4라는 값(요소) 추가

print(list_1) result: [1, 2, 3, 4, 5]

추가 = + 연산자 연결과 유사 (차이점: 연결시키면서 원본의 변화가 일어난다는 '파괴적 처리')

list_1.extend(list_2) → **extend**는 리스트와 리스트의 연결을 담당

print(list_1) result: [1, 2, 3, 4, 5, 4, 5, 6]

비파괴적 처리의 연산자는 사용 결과가 언제나 동일하지만,

파괴적 처리의 연산자는 사용결과가 달라진다. (예: append(5)를 list_1에 집어 넣었을 때, 첫번째는 1, 2, 3, 5가 되지만 계속 실행시키면 뒤에 5가 추가적으로 들어가게 된다 = 파괴적 처리의 특징

▸ 리스트 연산자: 요소 제거

※ 삭제 관련된 함수 헷갈리지 않게 정리해두자**! :** 공통점은 모두 '파괴적 처리'가 이루어진다는 점**~!**

a. clear( ) : 리스트의 모든 요소를 제거하여 빈 리스트로 만들고 원본 리스트 변경
b. pop( ) : 리스트에 지정된 인덱스 위치에 있는 요소를 제거하고 반환되며, 원본 리스트 변경
c. remove( ) : 리스트에서 지정된 값을 가진 첫 번째 요소를 제거하고 원본 리스트 변경
d. del( ) : 리스트에서 지정된 인덱스 위치에 있는 요소를 제거하고 원본 리스트 변경

※ **pop**과 **del**은 인덱스 기반으로 데이터가 삭제되며, **remove**는 값을 기반으로 삭제된다.

case 1. del(delete) 키워드 사용

del list_1[2]

print(list_1) reslut: [1, 5, 4, 6, 5, 4, 5, 6]

case 2. pop 사용

list_1.pop(1)

print(list_1) result: [1, 4, 6, 5, 4, 5, 6]

case 3. remove 함수를 사용하여 값으로 제거

list_1.remove(4)

print(list_1) result: [1, 6, 5, 4, 5, 6]

리스트 안의 모든 요소 제거 = 비우기(리스트 구조는 살아있다.)

list_1.clear() print(list_1) ; result: []

→ 식별자는 살아있다 ; why? list_1 자체가 사라졌다면 실행결과에서 에러가 발생하였을 것!

▸ 리스트 정렬 : **.sort(reverse = True / False)**

오름차순

score = [75, 55, 40, 90, 100]

score.sort() default value = False = 오름차순; 오름차순의 경우 따로 값을 지정하지 않아도 된다.

score.sort(**reverse = False**) ※ 파이썬에서는 **True, False** 대소문자 구별 必

print(score) → result: [40, 55, 75, 90, 100]

# 내림차순
score.sort(**reverse = True**)
print(score) → result: [100, 90, 75, 55, 40]

▸ 전개 연산자

전개 연산자(*)는 반복 가능한 객체를 언패킹하는데 사용하는 특별한 연산자이다.
# case 1. 리스트 안에서 사용
list_a = [1, 2, 3] [1, 2, 3]
list_b = [* list_a] [1, 2, 3] → a의 요소를 하나씩 순차대로 집어 넣는다는 의미
list_c = [* list_a, * list_a] [1, 2, 3, 1, 2, 3]

case 2. 함수에서 사용
print(list_a) # 리스트로 확인하는 것 = 리스트 자체를 출력
print(* list_a) # 리스트 안에 있는 요소를 하나씩 꺼내서(=언패킹하여) 값을 반환 (따라서 개별 요소로 출력)
result: [1, 2, 3] / 1 2 3

12. 딕셔너리

▸ 단일 딕셔너리 생성하기
sd_dic = {"이름": "정대만", "나이":19, "키": 183, "몸무게":70} # 콜론(:)을 중심으로 앞은 키, 뒤는 값
sd_list = ["정대만", 19, 183, 70]
#똑같은 값을 가지고 있는 딕셔너리와 리스트 (형태의 차이)
이름 보기
print(sd_list[**0**]) result: 정대만
print(sd_dic['이름']) result: 정대만
→ 딕셔너리는 중괄호 안에 키와 값을 넣고, 출력시에는 키를 꺼내기 (=값이 출력)
※ 딕셔너리는 생성 시에는 중괄호를 사용하지만, 출력할 땐 리스트와 동일하게 대괄호 사용

▸ 값 변경하기
sd_dic['키'] = 185 # 내가 원하는 값으로 변경하여 입력
print(sd_dic['키'])
result: 185
cf. 리스트는 인덱스 번호 sd_list[2] = 185
※유의! print(sd_dic{}) → 값을 넣을 때만 중괄호, 출력시에는 리스트와 똑같이 대괄호!

▸ 여러 개의 값을 가지는 딕셔너리 생성
bs = {"선수" : ["채치수", "정대만", "송태섭", "강백화", "서태웅"], "감독": ["안감독"], "매니저":["한나"]}
# 감독이나 매니저는 따로 리스트로 작성하지 않아도 됨 why? 단일값이기 때문
print(bs)
result: {'선수': ['채치수', '정대만', '송태섭', '강백화', '서태웅'], '감독': ['안감독'], '매니저': ['한나']}
print(bs["선수"]) # 전체 값 result:['채치수', '정대만', '송태섭', '강백화', '서태웅']
print(bs["선수"][1]) # 개별 값 출력 result: 정대만

▸ 값 변경하기

```
# case 1.
bs['매니저'] = "소연"
print(bs['매니저'])
# case 2.
bs['매니저'][0] = "소연"
print(bs['매니저']
```

▸ 키 값 제거

del bs['상대팀'] → del 삭제하고자 하는 키 이름 작성

```
print(bs)
result: {'선수': ['채치수', '정대만', '송태섭', '강백화', '서태웅'], '감독': ['안감독'], '매니저': '소연', '별명': ['고릴라', '불꽃남자']}
※ ---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-86-9a0fca9ae5d8> in <cell line: 1>()
----> 1 del bs['상대팀']
      2 print(bs)
KeyError: '상대팀'  > 딕셔너리에 없는 키를 사용하면 발생
-> 이런 에러 문구가 뜬다면 삭제를 두 번 시행시킨 것!
```

13. 튜플

▸ 튜플 생성

※ 하나의 값만을 가진 튜플의 뒤에 ,를 붙이는 이유? 데이터 타입이 int가 아닌 튜플 인식해야 하기 때문이다.

```
list_a = [1, 2, 3]
tuple_a = (1, 2, 3)
print(list_a); print(tuple_a)  result  [1, 2, 3]   (1, 2, 3)
print(type(list_a)); print(type(tuple_a))   result  <class 'list'>   <class 'tuple'>
```

cf. 튜플은 소괄호 없이 생성이 가능하다.

```
tuple_a = 1, 2, 3
print(tuple_a); print(type(tuple_a))
(1, 2, 3) ; <class 'tuple'>
```

▸ 튜플의 값 가져오기

```
print(list_a[0]); print(list_a[1]); print(list_a[2])
print(tuple_a[0]);print(tuple_a[1]);print(tuple_a[2])
```

→ 둘의 출력물은 1, 2, 3으로 동일하다.

▸ 튜플의 값 변경하기?

```
list_a[0] = 10
print(list_a)     result [10, 2, 3]
tuple_a[0] = 4
```

print(tuple_a)
# 에러 발생 why? 튜플은 요소 변경 불가 (※ 요소 변경이 가능한 것은 리스트와 딕셔너리)
TypeError: 'tuple' object does not support item assignment

14. 복합자료형 셋

▸ 복합자료형 셋(set)의 특징?
리스트와 튜플은 인덱스, 즉 순서가 존재한다. 하지만 셋은 순서가 존재하지 않는 복합자료형이다.
셋을 사용하는 경우: 집합에 관련된 것을 쉽게 처리하기 위해 만든 자료형이다. (중복 허용 불가, 순서 無)

s1 = set([1, 2, 3]) → print(s1) => {1, 2, 3}
s3 = set("Hello")
print(s3)
result : {'H', 'l', 'e', 'o'}

▸ 셋 자료형에 인덱싱으로 접근하려면 리스트나 튜플로 변환해야 한다 (why? 셋은 순서 개념이 없으니!)

```
s1 = set([1, 2, 3])
l1 = list(s1)
print(l1)
print(l1[0])
t1 = tuple(s1)
print(t1)
print(t1[0])
```

▸ 셋을 사용하는 이유 : 집합형 자료를 처리할 때 유용
셋으로 집합연산하기
1. 합집합 : **& / .union(___)**

```
print(s1 & s2)
print(s1.union(s2))
```

2. 교집합 : **| / .intersection( )**

```
print(s1 | s2)
print(s1.intersection(s2))
# 3. 차집합 : - / .difference( )
print(s1 - s2)
print(s2 - s1)
print(s1.difference(s2))
print(s2.difference(s1))
```

※ 차집합은 앞뒤 순서 배치에 따라 결과가 달라지므로 유의

15. 반복문: for문과 while문

a. for문의 경우 원하는 횟수만큼 반복하는 경우

i. 범위를 지정하는 range 개념 익히기 (매개변수는 반드시 정수형태여야만 한다) → 실수 X
range(시작번호, 끝번호, 간격) ※ 유의 : range함수는 끝번호를 포함하지 않는다.

case 1) 끝 번호만 작성하는 경우 (시작번호 생략)

range(10) → 값을 하나만 작성할 경우는 끝번호만 지정하겠다는 의미

초기값을 별도로 지정하지 않을 경우 0부터 시작된다. (why? 파이썬은 제로 인덱스)

(예) for i in range(10): → 0 ~ 9까지 반복된다는 의미

case 2) 시작번호와 끝 번호를 작성하는 경우 (간격은 생략)

range(1, 10) → 값을 두개를 지정할 경우 시작과 끝을 지정한다는 의미

간격의 초기값을 1이므로 1부터 9까지의 숫자가 1의 간격으로 '모두' 출력

case 3) 시작번호와 끝번호, 간격을 작성하는 경우

range(1, 10, 2) → 1부터 9까지 2의 간격으로 출력(1, 3, 5, 7, 9)

ii. 리스트 반복문

(a) 단일 리스트 = 1차원 리스트

```
list_a = [1, 2, 3]
for i in list_a:
  print(i) # 3개의 요소를 가지고 있어서 3번 반복   → result: 1 / 2 / 3
```

(b) 다중 리스트 = 2차원 리스트

```
list_b = [[1, 2, 3], [4, 5, 6]] # 2차원 리스트
for i in list_b:
  print(i) # 2개의 요소를 가지고 있어서 2번 반복; 문제점: 여전히 리스트 형식으로 출력되고 있다.
```

= 리스트 안에 리스트가 있다는 말이므로 또 한 번의 반복이 이루어질 수 있다.

> result: [1, 2, 3] / [4, 5, 6] ; 여전히 리스트 형태

```
for i in list_b: # 처음에 반복할 때는 [1, 2, 3] 과 [4, 5, 6]을 꺼내고,
  for j in i: # 그 i 안에 [1, 2, 3]이 있는 걸 하나씩 꺼낸다 = 1차원 리스트 값을 꺼내는 것과 같음
    print(j) # 최종적으로 1차원 리스트로 쪼갠 그 j를 꺼내보아라 = 값이 하나씩 출력되게 된다.
```

> result: 1 / 2 / 3 / 4/ 5/ 6

결론: n차원 리스트의 경우에는 1차원 리스트의 형태가 될 때까지 반복한 후 출력

▸ 1차원의 데이터와 2차원의 데이터가 함께 있는 경우?!

```
# list_c = [[1, 2, 3], [4, 5, 6], 7]
# for i in list_c:
#   print(i) # 여기까지는 정상 출력> 결과: [1, 2, 3] / [4, 5, 6] / 7
#   for j in i:
#     print(j) # 7은 리스트가 아니기 때문에 개별값을 개별값으로 꺼낸다는 것은 불가능
#     해결방법? 리스트인 값과 리스트가 아닌 값을 구분하는 '조건'을 걸어서 반복문을 돌리기
# 에러 문구: TypeError: 'int' object is not iterable
```

for i in list_c:

print(type(i)) # 각 요소의 데이터 타입 확인

유의점: 이때 for 반복문 없이 print(type(i))로만 할 경우, class 'list'로 나온다

→ why? 리스트 안 개별값이 아닌 리스트 전체로 인식하여 값을 출력하기 때문

∴ 리스트 안에 개별값을 확인하고 싶을 땐, 반복문으로 각 요소를 쪼갠 후 데이터 타입을 확인

결과값

<class 'list'> / <class 'list'> / <class 'int'>

```
for i in list_c:
 if type(i) ==  list:
   for j in i:
     print(j)
 else:
     print(i)
```

iii. 리스트와 범위로 반복문

when? 인덱스 번호와 값을 동시에 출력하고 싶을 때

list_a = [21, 34, 11, 5, 67]

print(len(list_a)) 5

print(range(len(list_a))) = range(0, 5)

print(list(range(len(list_a)))) [0, 1, 2, 3, 4]

```
for i in range(len(list_a)):
 print("인덱스", i,"값", list_a[i])
```

result

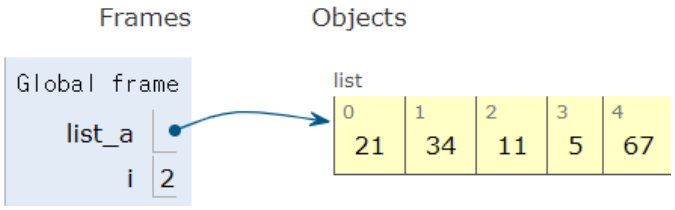
인덱스 0 값 21

인덱스 1 값 34

인덱스 2 값 11

인덱스 3 값 5

인덱스 4 값 67



iv. 딕셔너리 반복문

▸ 단일 딕셔너리 반복문

sd_dic = {"이름": "정대만", "나이":19, "키": 183, "몸무게":70}

for i in sd_dic:

 print(sd_dic[i])

result: 정대만 / 19 / 183 / 70

▸ 다중 딕셔너리 반복문

```
character = {"name" : "정대만", "number" : 14, "records" : {"three_point" : 24, "rebound" : 1}, "schools"
: ["무석중", "북산고"]}
for i in character:
  print(i)
```

result → key값이 출력된다 : name / number / records / schools

<values를 출력하고 싶다면?> → result → 정대만 / 14 / 24 / 1 / 무석중 / 북산고

```
for i in character:
  if type(character[i]) == dict:
    for j in character[i]:
      print(character[i][j])
  elif type(character[i]) == list:
    for k in character[i]:
      print(k)
  else:
    print(character[i])
```

v. 역반복문

(1) 음수를 활용하여 역반복문 만들기

```
for i in range(5, 0 , -1):
 print(i)
```

result → 5 ~ 1 역순으로 출력

```
for i in range(5, -1, -2):
print(i)
```

result → 5 / 3 / 1

(2) reversed( )를 사용하여 역반복문 만들기

```
for i in reversed(range(1, 6)):
  print(i)
```

result → 5 ~ 1 역순으로 출력

b. while문은 주어진 조건이 **'True'**일 때만 실행하는 경우

for문 : **for i in** 반복횟수 **…**

반복횟수 안에는 리스트, 딕셔너리, 문자열, 범위(range) 등

```
i = 0
while i < 10:
  print(i)
  i += 1
```

result: 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

▸ 값이 있는 동안 반복

```
t = [1, 2, 3, 2, 4, 2, 5]
value = 2
while value in t:
  t.remove(value)
print(t)
result: [1, 3, 4, 5]
```

▸ 반복을 종료하기 : break → 현재의 코드 진행을 무시하고, 즉시 반복문을 종료하고 싶을 때 사용

```
i = 0
while True:
  print(i)
  i += 1
  yn = input("종료할까요? (종료를 원하시면 Y나 y를 입력하세요): ")
  if yn in ["Y", "y"]:
    print("반복 종료")
    break (Y나 y가 아닌 값을 입력했을 경우에는 while문이 계속 실행)
```

▸ 현재의 반복을 생략하고 다음 반복 진행하기 : continue

```
list_a = [100, 30, 150, 77, 258]
for i in list_a:
  if i < 100:
    continue
  print(i)  → 100 /  150  / 258
```

※ 반복문에서 break와 continue의 차이?

영어단어의 어감으로만 봐도 직관적인 차이를 발견할 수 있다. break는 무언가 끝내는 느낌, continue는 지속되는 느낌을 내포하고 있기 때문이다. 실제로 반복문에서 **break**를 사용하면 반복문을 완전히 종료하고 반복문 바로 다음의 코드로 이동한다. 반면에 **continue**는 반복문 내에서 사용될 때**,** 현재의 반복을 중단하고 반복문의 다음 반복 단계로 넘어간다.

∴ break는 반복문 자체를 종료하고, 반복문이 아닌 그 다음의 코드를 실행

continue는 현재의 반복은 중단하고 다음 반복 단계로 넘어가는 즉, 반복문 자체가 중단된 것은 아닌 상태

▸ 반복문에서 인덱스 번호 부여로 유용한 enumerate( )

enumerate는 파이썬 내장 함수로, 반복 가능한(iterable) 객체(예: 리스트, 튜플, 문자열 등)를 입력받아 인덱스와 해당 요소로 구성된 이터레이터(iterator)를 반환한다. 특히 반복문에서 인덱스와 요소를 동시에 사용해야 할 때 유용하게 활용된다.

기본 구문 : enumerate(iterable, start=0)

```
fruits = ['apple', 'banana', 'orange']
for index, fruit in enumerate(fruits):
    print(index, fruit)

result
0 apple
1 banana
2 orange
```

16. 함수

함수는 재사용 가능한 코드 블록으로, 특정 작업을 수행하기 위한 '독립적인 기능' 단위이다. 함수를 정의하기 위해서는 def 키워드를 사용하고 함수의 이름과 매개변수(parameter)를 지정한다.

```
def add(a, b):
    result = a + b
    return result
```

▸ 매개변수 종류 : 작성 순서) 위치 = 일반 → 가변 → 기본

(1) 위치 매개변수 : 함수 정의에서 매개변수의 순서대로 인수가 전달되는 방식 ≒ format( )과 비슷

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")
greet("Alice", 25)
```

(2) 기본값 매개변수 : 함수 정의 시 매개변수에 기본값을 할당하는 방식 (※ 매개변수 목록 뒤쪽 위치)

```
def greet(name, age=30):
    print(f"Hello, {name}! You are {age} years old.")
greet("Alice")  # age 기본값(30)이 사용됨 → 위치 매개변수보다 뒤에 위치
greet("Bob", 35)  # 인수로 전달된 값(35)이 사용됨
```

(3) 키워드 매개변수 : 인수를 매개변수 이름과 함께 전달하는 방식

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")
greet(age=25, name="Alice")  # 키워드 매개변수 사용
```

→ 키워드 매개변수가 위치와 기본값과 비슷해보이지만, 다른점은 변수를 지정했다는 것!
즉, 위의 두 변수는 "무엇은 무엇"이 아니라 값만 넣었다면, 키워드 매개변수는 "무엇"은 무엇이라 지정하였기에 위의 두 매개변수와 달리 순서를 지키지 않아도 구동된다. 왜? 키워드 중심이므로!

(4) 가변 매개변수 : 함수 정의에서 개수가 정해지지 않은 인수를 처리하는 방식 (하나만 사용 가능)
위의 세 매개변수는 몇 개의 변수가 들어올지 이미 정해진 것이지만, 가변 매개변수의 경우에는 몇 개의 변수가 입력될지 모를 때 사용 (예를 들어 A 데이터는 3개가, B 데이터는 2개가 들어올 때 이 모든 데이터를 예외없이 수용하고 싶을 때 가변 매개변수를 사용 → 나는 너희를 다 받아줄게)

*변수명 형태로 매개변수를 정의하고, 여러 개의 인수를 튜플로 묶어서 전달

```
def 매개변수(ex1, ex2, *가변매개변수):
    print(ex1)
    print(ex2)
    print(가변매개변수)
매개변수(1, 2, 3, 4, 5, 6, 7)
```

```
ftn1(1,2,3)
1
2
(3,)
ftn2(1,2,3)
1
(2, 3)
1
```

```
def ftn1(a, b=1, *c):
    print(a)
    print(b)
    print(c)

def ftn2(a, *b, c=1):
    print(a)
    print(b)
    print(c)
```

(5) 반환값 return (return을 만나면 무조건 함수 호출이 종료된다)

13. 함수

▸ 사용자 정의 함수

▸ 리턴 : 최종적으로 보여줄 값을 작성할 때 용이

```
# 리턴 = 반환값 (∴ return을 만나면 무조건 함수 호출이 종료된다.)
# 리턴값 = 함수 적용 결과
# case 1. 자료 없이 리턴 → 별도의 출력물 없이 종료하게 된다.
def return_test():
  print("A")
  return # return을 만나면 함수는 끝나게 된다.
  print("B")

return_test()

# case 2. 자료와 함께 리턴
def return_test():
```

```
  return "B" # 뒤에 값이 있으면 출력하고 함수 종료
return_test()
```

▸ 입력한 값을 모두 더하는 함수

```
# 함수 정의하기
def sum_all(start, end):
  output = 0
  for i in range(start, end+1):
    output += i
  return output
# 일반 매개변수로 사용하기 = 순서대로 입력해야 한다는 의미
sum_all(1,3)


# 키워드 매개변수로 사용 = 순서 상관 없음
sum_all(start = 1, end = 3)
결과는 모두 동일하게 6으로 출력
# 기본매개변수
# 간격을 추가해서 합계
def sum_all(start = 1, end = 10, step = 2):
  output = 0
  for i in range(start, end+1, step):
    output += i
  return output
sum_all ()  출력값: 25


# 일반매개변수로 사용 = 순서대로 입력
sum_all(1, 5, 2)  출력값: 9
# 키워드 매개변수로 사용 = 순서 상관 없음
sum_all(start = 1, end = 5, step = 2)  출력값: 9
# 키워드 매개변수 일부만 사용
sum_all(start = 1, step = 2)  출력값: 25
```

※ 함수의 매개변수의 작성 순서 : 일반 → 가변 → 기본

▸ 재귀 함수

```python
# factorial
# 반복문으로 구현
def factorial(n):
  output = 1
  for i in range(1, n+1):
    output *= i
  return output
print(factorial(1))   result: 1
print(factorial(2))   result: 2
print(factorial(3))   result: 6


def factorial(n):
  output = 1
  for i in range(n, 1-1, -1): # order of factorial
    output *= i
  return output
print(factorial(1))   result: 1
print(factorial(2))   result: 2
print(factorial(3))   result: 6
# 재귀함수로 구현
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
    # n = 3
    # 3 * factorial(2)
    # 3 * 2 * factorial(1)
    # 3 * 2 * 1 * factorial(0)
    # 3 * 2 * 1* 1 = 6
```

▸ 피보나치 수열

```
def fibonacci(n):
  if n == 1:
    return 1
  elif n == 2:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
    # n = 3
    # f(3) = f(2) + f(1) = 1 + 1+ = 2
print(fibonacci(1))   result 1
print(fibonacci(2))   result 1
print(fibonacci(3))   result 2
print(fibonacci(4))   result 3
```

▸ 키워드 global

```
# 재귀 함수 사용 횟수 계산
count = 0
def fibonacci(n):
  global count # 함수 밖에 있는 count라는 변수를 가져온다는 의미 (있어야 오류가 발생하지 않음)
  count += 1
  if n == 1:
    return 1
  elif n == 2:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(25))     result 75025
print(count)             result 150049
```

▸ 메모화

```
f_dict = {
    1: 1, 2: 1
```

```
}
# 조기 리턴
def fibonacci(n): # n은 딕셔너리 키
  if n in f_dict:
    return f_dict[n] # 존재하는 키의 값을 가져오는 경우 = 계산한 것이 있다.
  output = fibonacci( n-1) + fibonacci(n-2)
  f_dict[n] = output # 존재하지 않는 키의 값을 정하는 경우 = 계산한 것이 없다.
  return output


  print(fibonacci(50))   result 12586269025


# 조기 리턴을 사용하지 않는 경우
f_dict =(n): # n은 딕셔너리 키
  if n in f_dict:
    return f_dict[n]
  else:
    output = fibonacci(n-1) + fibonacci(n-2)
    f_dict[n] = output
    return output
```

▸ 리스트 평탄화

```
# 2차원 리스트 반복문 2번 사용해서 평탄화
# 실제로 중과로가 몇 번 나올지 예상할 수 없다.
# 이럴 땐 재귀함수로 해결할 수 있다.


def flat(data):
  output = []
  for i in data: # 반복
    if type(i) == list: # 리스트인 경우
      output += flat(i) # 리스트가 아닐 때까지 반복
    else: # 요소인 경우
      output.append(i) # 요소 추가 -> 개별값만 추가할 때는 append, 리스트 전체를 추가할 땐 extend 따라서
여기서 extend를 사용할 수 없다.
  return output
```

```
# [1, 2, 3] -> flat([1, 2, 3]) → 리스트 아니므로 요소 추가
# [4, [5, 6]] -> flat([4, [5, 6]]) → 4는 요소 추가
# [5, 6] -> flat([5, 6]) → 리스트 아니므로 요소 추가
# 7 → 리스트 아니므로 요소 추가
# [8, 9] -> flat([8, 9]) → 리스트 아니므로 요소 추가
a = [[1, 2, 3], [4, [5, 6]], 7, [8, 9]]
print(a)       result [[1, 2, 3], [4, [5, 6]], 7, [8, 9]]
print(flat(a)) result [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

▸ 몫 & 나머지

```
a, b = 63, 30
print(a // 10)   -> 6          print(a % b)    -> 3
# 몫과 나머지 값을 튜플 형태로 반환하는 함수
c, d = divmod(a, b)
print(c); print(d)   -> 0 ; 1
```

▸람다

```
# 함수의 매개변수에 사용하는 함수 = 콜백함수
def repeat_10(x):
  for i in range(10):
    x() # 함수 사용 = print_hi
def print_hi():
  print("hi")
repeat_10(print_hi) # print_hi = 콜백함수
# 이런 식으로 한번만 사용할 함수는 '람다'를 사용하면 된다.
hi / hi / hi / hi / hi / hi / hi / hi / hi / hi


# map 함수 = 콜백 함수 적용 결과 값
def sqr(x):
  return x ** 2
list_a = [1, 2, 3]
output_map = map(sqr, list_a) # 객체로 저장된 값이 출력되기에 output_map이라는 값에 저장
```

```
print(output_map)
print(list(output_map))
<map object at 0x7f9a7c10d7e0>
[1, 4, 9]



# filter 함수 = 콜백 함수 적용 결과 중에서 True인 것만 출력
def under_3(x):
  return x < 3
list_a = [1, 2, 3]
output_filter = filter(under_3, list_a) # 객체로 저장된 값이 출력되기에 output_map이라는 값에 저장
print(output_filter)
print(list(output_filter))
<filter object at 0x7f9a7c10f070>
[1, 2]


# 람다로 변경
list_a = [1, 2, 3]
# 콜백 함수 자리에 람다
# 람다 사용법: lambda 매개변수: return_value(기능 결과)
output_map = map(lambda x: x **2, list_a)
print(output_map)
print(list(output_map))


<map object at 0x7f9a7c134ac0>
[1, 4, 9]
list_a = [1, 2, 3]
output_filter = filter(lambda x: x < 3, list_a)
print(output_filter)
print(list(output_filter))
<filter object at 0x7f9a7c1354e0>
[1, 2]
```

▸ 리스트에서 사용했던 함수를 딕셔너리에서 사용하는 방법

```
# 딕셔너리 형태의 요소를 가지고 있는 리스트
players = [{"이름" : "강백호",
            "신장" : 189},
           {"이름": "서태웅",
            "신장" : 188},
           {"이름" : "채치수",
            "신장" : 195}]

def height(players):
  return players["신장"]
print(min(players, key = height)) # 키와 값을 모두 가지고 있는 딕셔너리의 경우 최소값을 찾는 방법
print(max(players, key = height))

result
{'이름': '서태웅', '신장': 188}
{'이름': '채치수', '신장': 195}
```

vs. list

```
# list의 경우 최대/최소값을 찾는 방법
player_list = [189, 188, 195]
print(min(player_list))
print(max(player_list))

result
188
195
```

<함수의 활용>

```
# 딕셔너리 형태의 요소를 가지고 있는 리스트
players = [{"이름" : "강백호",
            "신장" : 189},
           {"이름": "서태웅",
```

```
          "신장" : 188},
         {"이름" : "채치수",
          "신장" : 195}]

def height(i): # i가 딕셔너리 players
  return i["신장"]
# min/max함수 안에서
# key에서 사용하고 있는 함수의 매개변수로 딕셔너리 players 사용
print(min(players, key = height)) # 키와 값을 모두 가지고 있는 딕셔너리의 경우 최소값을 찾는 방법
print(max(players, key = height))
# 람다로 변경
print(min(players, key = lambda x : x["신장"]))
print(max(players, key = lambda x : x["신장"]))
print(min(players, key = lambda player : player["신장"])) # 실무에서는 전체를 복수형 값을 하나씩 꺼낼 때
단수형태로 작성하는 경우가 多
print(max(players, key = lambda player : player["신장"]))
```

▸ 정렬

```
players = [{"이름" : "강백호",
          "신장" : 189},
         {"이름": "서태웅",
          "신장" : 188},
         {"이름" : "채치수",
          "신장" : 195}]

players.sort(key = lambda x : x["신장"]) # 오름차순으로 정렬
print(players)
[{'이름': '서태웅', '신장': 188}, {'이름': '강백호', '신장': 189}, {'이름': '채치수', '신장': 195}]

players = [{"이름" : "강백호",
          "신장" : 189},
         {"이름": "서태웅",
          "신장" : 188},
```

```
        {"이름" : "채치수",
         "신장" : 195}]
players.sort(key = lambda x : x["신장"], reverse = True) # 내림차순으로 정렬
print(players)
```

```
# case 4. items () with dictionary
# 키와 값을 동시에 출력
sd_dic = {"이름": "정대만", "나이":19, "키": 183, "몸무게":70}
sd_dic.items() # 튜플형태
for i, j in sd_dic.items():
  print("key", i, "값", j)
# enumerate와 itesm 함수는 사용형식이 다르므로 유의!
key 이름 값 정대만 / key 나이 값 19 / key 키 값 183 / key 몸무게 값 70
리스트 내포

조건문이 추가된 리스트 내포
# 리스트 내포에 조건문 추가
# 과일만 리스트로 저장
list_a = ["사과", "참외", "수박", "토마토"]
fruit = [i # 표현식 (어떤 값을 집어넣을 것인가?)
        for i in list_a # 반복문
        if i != "토마토"] # 조건문
print(fruit)
result: ['사과', '참외', '수박']

여러 줄의 문자열 출력 들여쓰기 문제 해결하기
#문제 상황 제시
# 홀수, 짝수 구분하는 프로그램
number = int(input("정수를 입력하세요: "))
if number % 2 == 0: # 짝수
  print("""\
```

```
  입력한 숫자는 {}입니다.
  {}는 짝수입니다.\
  """.format(number, number))
else: # 홀수
  print("""\
  입력한 숫자는 {}입니다.
  {}는 홀수입니다.\
  """.format(number, number))


# 들여쓰기가 되는 문제가 발생


#문제 상황 해결
# 문제 해결 방법 case 1. → 실제로는 사용하지 않는다.
number = int(input("정수를 입력하세요: "))
if number % 2 == 0: # 짝수
  print("""\
입력한 숫자는 {}입니다.
{}는 짝수입니다.\
  """.format(number, number))
else: # 홀수
  print("""\
입력한 숫자는 {}입니다.
{}는 홀수입니다.\
  """.format(number, number))
# if~else 구문 안에서는 무조건 들여쓰기가 이루어져야 한다.
# 이 코드의 문제점은 일관성이 없어 '가독성'이 떨어진다는 문제가 발생한다.


# 문제 해결 방법 case 2.
number = int(input("정수를 입력하세요: "))
if number % 2 == 0: # 짝수
  print("입력한 숫자는 {}입니다.\n{}는 짝수입니다.".format(number, number))
else: # 홀수
  print("입력한 숫자는 {}입니다.\n{}는 홀수입니다.".format(number, number))
```

```
하지만, 문자가 길어질 경우 case 2를 사용하는 것이 효율적일까? NO!
이때 사용하는 것이 함수 join( )
step1. join () 함수의 사용법 살펴보기
print("-".join(["010","1234","5678"]))  result: 010-1234-5678
print(" ".join(["010","1234","5678"]))  result: 010 1234 5678
print("|".join(["010","1234","5678"]))  result: 010|1234|5678

위의 문제를 join 함수로 풀어본다면?
number = int(input("정수를 입력하세요: "))
if number % 2 == 0: # 짝수
  print("\n".join(["입력한 숫자는 {}입니다.","{}는 짝수입니다."]).format(number, number))
else: # 홀수
  print("\n".join(["입력한 숫자는 {}입니다.","{}는 홀수입니다."]).format(number, number))

result:
정수를 입력하세요: 24
입력한 숫자는 24입니다.
24는 짝수입니다.
# 소괄호를 사용한 문자 연결
a = (
   "나는 "
   "여름을 "
   "좋아한다."
)

print(a)       result 나는 여름을 좋아한다.
print(type(a))  result <class 'str'>
number = int(input("정수를 입력하세요: "))
if number % 2 == 0: # 짝수
  print(("입력한 숫자는 {}입니다.\n"
       "{}는 짝수입니다.").format(number, number))
else: # 홀수
```

```
 print(("입력한 숫자는 {}입니다.\n"
        "{}는 홀수입니다.").format(number, number))


본인의 취향과 입맛에 따라 사용하면 된다.
```

▸ 텍스트 파일 처리

```
# 파일 열기
file = open('base.txt', "w") # 원하는 경로를 복사하여 붙여넣기
# 텍스트 쓰기
file.write("Hello, Python?")
# 파일 닫기
file.close()
# 파일을 열고 닫는 과정에 코드가 많이 생기면 파일을 닫지 않는 실수가 발생
# 이를 방지하기 위해서 with를 사용
with open('base.txt.', "w") as file:
  file.write("Hello, Python?!")
# with문이 끝나면 자동으로 파일이 닫힘
```

▸텍스트 파일 한줄씩 읽기

```
# 이름, 키, 몸무게 → csv 텍스트 파일로 저장
import random
hangul = list("가나다라마바사아자차카타파하")
with open ('info.txt',"w") as file:
  for i in range(1000):
    name = random.choice(hangul) +  random.choice(hangul) + random.choice(hangul)
    height = random.randrange(140,200)
    weight = random.randrange(40,150)
    file.write("{},{},{}\n".format(name, height, weight))
```

```
# 텍스트 파일 한줄씩 읽기 & 새로운 변수 생성 및 출력
with open('info.txt', "r") as file:
  for line in file:
```

```
name, height, weight = line.strip().split(", ")
if (not name) or (not height) or (not weight):
  continue
# 체중(kg) / 키(cm) 제곱
bmi = int(weight) / (int(height) / 10) **2
result = ""
# 25 보다 크거나 같으면 과체중
# 18.5 보다 크거나 같으면 정상
# 나머지는 저체중

if bmi >= 25:
  result = "과체중"
elif bmi >= 18.5:
  result = "정상"
else:
  result = "저체중"
# print("이름 : {}, 키:{}, 몸무게: {},BMI:{}, 결과:{}".format(name, height, weight,bmi,result))
print("\n".join(["이름 : {}", "키:{}", "몸무게: {}","BMI:{}", "결과:{}"]).format(name, height, weight, bmi, result))
```

14. 구문 오류와 예외

▸ 오류의 종류

a. 구문 오류

프로그램 실행 '전'에 발생하는 오류 (예: 오타)

```
# 일반적인 상황 : 오타
# 프로그램 실행 전에 발생하는 오류
print("Good Morning)
# SyntaxError : 이 문구가 나왔을 땐 오타가 발생한 것이 없는지 찬찬히 살펴보기
```

b. 예외 / 런타임 오류

프로그램 실행 '중'에 발생하는 오류 (예외 처리)

```
# (= 런타임 오류), 프로그램 실행 중에 발생하는 오류
list_z[0]
# NameError: name 'list_z' is not defined
# 오류 해결 방법
```

```
list_z = [1, 2]

list_z[0]

result : 1
```

▸ 예외 상황 만들기

case 1. isdigit() 함수를 사용하여 T/F확인 ; if 조건문 사용

True의 경우일 때 내가 실제 구동하고 싶은 코딩을 작성

False의 경우 오류처리가 나지 않도록 예외 문장 작성하기

```
# 원의 반지름을 입력해서 원의 둘레와 넓이 계산해주는 프로그램

r = int(input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")) # int 이외의 값이 들어왔을 때 어떻게 예외

처리를 할 것인가?

# 정수 입력이라는 조건을 걸었음에도 실수로 입력하는 경우가 있을 것 -> 이것이 바로 오류 상황 발생

# 실수나 문자를 포함한 숫자를 입력해서 예외 상황 만들기

pi = 3.14

print("원의 반지름", r)

print("원의 둘레", 2 * pi * r)

print("원의 넓이", pi * r * r)


원의 반지름을 입력하세요.(단, 정수로 입력할 것) 5

원의 반지름 5

원의 둘레 31.400000000000002

원의 넓이 78.5


if) 5.0을 입력한다면? → 에러 발생!

에러가 발생되지 않고 프로그램을 구동시킬 수는 없을까?

# 조건문으로 예외 처리 → 오류가 발생되지 않고 프로그램이 정상적으로 종료될 수 있도록 한다,

r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")

pi = 3.14

if r.isdigit(): # 숫자로 변환할 수 있는 경우

  r = int(r)

  print("원의 반지름", r)

  print("원의 둘레", 2 * pi * r)

  print("원의 넓이", pi * r * r)

else:
```

```
 print("정수를 입력해주시겠어요?")
```

```
원의 반지름을 입력하세요.(단, 정수로 입력할 것) 2.36
정수를 입력해주시겠어요?
```

case 2. try 구문 사용

try ~ except (if ~ else 구문과 비슷한 사용법)

try 구문은 프로그래밍 언어의 구조적 문제로 인해 조건문만으로 예외를 처리할 수 없는 경우에 사용하며, 어떤 상황에 예외가 발생하는지 완벽하게 이해하고 있지 않아도 프로그램이 강제로 죽어버리는 상황을 막을 수 있다.

```
# try ~ except 구문
try:
 r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")
 pi = 3.14
 r = int(r)
 print("원의 반지름", r)
 print("원의 둘레", 2 * pi * r)
 print("원의 넓이", pi * r * r)
except:
 print("정수를 입력해주시겠어요?")
```

```
원의 반지름을 입력하세요.(단, 정수로 입력할 것) 5.2
정수를 입력해주시겠어요?
```

예외가 발생하면 일단 처리해야 하지만, 해당 코드가 딱히 중요한 부분이 아닌 경우 프로그램 강제 종료부터 막는 목적으로 except 구문에 아무것도 넣지 않고 try 구문 사용

```
# pass 사용 가능
try:
 r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")
 pi = 3.14
 r = int(r)
 print("원의 반지름", r)
 print("원의 둘레", 2 * pi * r)
 print("원의 넓이", pi * r * r)
except:
 pass
```

try ~ exception ~ else 구문

try excep 구문 뒤에 else 구문 붙여 사용하면 예외가 발생하지 않았을 때 실행할 코드 지정 **可**

이때, 예외 발생 가능성 있는 코드만 try 구문 내부에 넣고 나머지는 모두 else 구문으로 빼는 경우가 많다.

```
# try ~ except ~ else 구문
try:
  r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")
  pi = 3.14
  r = int(r)
except: # 예외가 발생할 경우
  print("정수를 입력해주시겠어요?")
else: # 정상적으로 프로그램이 구동될 경우
  print("원의 반지름", r)
  print("원의 둘레", 2 * pi * r)
  print("원의 넓이", pi * r * r)

case 1. except 실행
원의 반지름을 입력하세요.(단, 정수로 입력할 것) 5.11
정수를 입력해주시겠어요?

case 2. else 실행
원의 반지름을 입력하세요.(단, 정수로 입력할 것) 4
원의 반지름 4
원의 둘레 25.12
원의 넓이 50.24
```

finally 구문

예외 처리 구문에서 가장 마지막에 사용할 수 있는 구문으로 예외 발생 여부와 관계없이 '무조건 실행'할 경우 사용

```
# try ~ except ~ else ~ finally 구문
try: # 예외 발생할 수 있는 코드 입력
  r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")
  pi = 3.14
  r = int(r)
except: # 예외가 발생할 경우 처리할 코드
```

```
  print("정수를 입력해주시겠어요?")
else: # 정상적으로 프로그램이 구동될 경우 처리할 코드
  print("원의 반지름", r)
  print("원의 둘레", 2 * pi * r)
  print("원의 넓이", pi * r * r)
finally:
  print("하여튼 프로그램이 정상적으로 종료되었어요.")
```

else (정상 구현)

원의 반지름을 입력하세요.(단, 정수로 입력할 것) 5

원의 반지름 5

원의 둘레 31.400000000000002

원의 넓이 78.5

하여튼 프로그램이 정상적으로 종료되었어요.

except (오류 발생)

원의 반지름을 입력하세요.(단, 정수로 입력할 것) 5.2

정수를 입력해주시겠어요?

하여튼 프로그램이 정상적으로 종료되었어요.

finally는 정상 구동 여부와 관계없이 무조건 실행되는 코드

★ **finally**에 대한 오해

finally 키워드 설명 예제로 '파일 처리'를 자주 사용하나, 실제 finally의 사용과는 전혀 무관

파일이 제대로 닫혔는지 확인하기 위해서는 파일 객체의 closed 속성으로 알 수 있다.

※ try, except, finally 구문의 조합

try 구문은 단독으로 사용할 수 없으며**,** 반드시 **except** 또는 **finally** 구문과 함께 사용

else 구문은 반드시 **except** 구문 뒤에 사용해야 한다.

a. try + except 구문 조합
b. try + except + else 구문 조합
c. try + except + finally 구문 조합
d. try + except + else + finally 구문 조합
e. try + finally 구문 조합

이 외의 조합은 구문 오류가 발생한다.

try 구문 내부에서 **return** 키워드를 사용하는 경우

- try 구문 내부에 return 키워드에 있다.
- try 구문 중간에서 탈출해도 finally 구문은 무조건 실행된다.
- 함수 내부에서 파일 처리 코드를 깔끔하게 만들고 싶을 때 finally 구문을 활용하는 경우가 많다.

반복문과 함께 사용하는 경우: break 키워드로 try 구문 전체 빠져나가도 finally 구문 실행

▸ 예외 객체: 예외 발생 시 예외 정보가 저장되는 곳

클래스 exception: 처음 예외 객체를 사용하다보면 예외의 종류를 몰라 당황하는 경우 **有**, 모든 예외의 어머니

```
# 예외 정보를 저장하는 객체
try:
  r = input("원의 반지름을 입력하세요.(단, 정수로 입력할 것) ")
  pi = 3.14
  r = int(r)
  print("원의 반지름", r)
  print("원의 둘레", 2 * pi * r)
  print("원의 넓이", pi * r * r)
except Exception as exception: # 예외객체 생성
  print("예외 종류: ", type(exception))
  print("예외 객체: ", exception)
result: 원의 반지름을 입력하세요.(단, 정수로 입력할 것) 2..
예외 종류:  <class 'ValueError'>
예외 객체:  invalid literal for int() with base 10: '2..'



# 여러가지 예외 상황
a = [1, 2, 3, 4, 5]
try:
# 첫 번째 예외 상황: 정수가 아닌 문자열을 입력했을 때
  number = int(input("정수를 입력하세요: "))

# 두 번째 예외 상황:인덱스에 없는 정수 번호를 입력한 경우
  print("{}번째 요소는 {}입니다.".format(number, a[number]))
except Exception as exception:
  print("예외 종류: ", type(exception))
```

```
 print("예외 객체: ", exception)


result
정수를 입력하세요: 24
예외 종류:  <class 'IndexError'>
예외 객체:  list index out of range
```

15. 모듈

- 여러 변수와 함수를 가지고 있는 집합체로 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만든 파이썬 파일
- 표준모듈: 파이썬 기본 내장 모듈
- 외부모듈: 다른 사람이 만들어서 공개한 모듈
- 모듈 사용 **: import** 키워드 사용

from 모듈 이름 import 사용할 변수나 함수

이때 모든 변수나 함수를 가져오고 싶다면 *를 사용, 여러개를 사용할 경우 (,)

```
# 내장모듈
import math
math.sin(1)
0.8414709848078965
import math as m
m.sin(1)
from math import sin
sin(1)
0.8414709848078965
random.uniform(10,20) # 10 ~ 20 사이에 있는 실수를 뿌려라
result: 14.756655655588556
a = [1, 2, 3, 4, 5]
random.shuffle(a)
print(a)
result: [5, 3, 1, 4, 2]
```

▸ 현재 날짜 시간: 모듈 불러오기

```
# 모듈 부르기: import + 모듈 이름
import datetime
```

```
now = datetime.datetime.now()
print(now) # 구글 본사 위치의 시간으로 표기되기에 우리나라 시간과 맞지 않음
print(now.year)
print(now.month)
print(now.day)
print(now.hour)
print(now.minute)
print(now.second)
print(now.weekday) # 결과값이 오브젝트 형태로 나온다.
→ 출력값
<built-in method weekday of datetime.datetime object at 0x7f7b6e825740>
print(now.weekday()) # 괄호가 들어가야 우리가 원하는 형식으로 나온다.
# 이때 결과 값이 숫자 형식으로 나오는데, 0;월요일 ~ 6; 일요일
```

배운 내용 응용하기

```
# format () 함수를 이용
import datetime
now = datetime.datetime.now()
"{}년 {}월 {}일 {}시 {}분 {}초".format(now.year, now.month, now.day, now.hour, now.minute, now.second)
```

```
# 계절을 알려주는 프로그램
import datetime
now = datetime.datetime.now()
if 3 <= now.month <= 5:
  print("봄")
if 6 <= now.month <= 8:
  print("여름")
if 9 <= now.month <= 11:
  print("가을")
if now.month == 1 or 1 <= now.month <= 2: #사고 중요
  print("겨울")
# 오전 오후를 구별하는 프로그램
import datetime
now = datetime.datetime.now()
```

```
korea_hour = now.hour + 9
if korea_hour <  12:
  print("오전")
if korea_hour >= 12:
  print("오후")
```

```
# 오전 오후 구별 프로그램 + format()
import datetime
now = datetime.datetime.now()
korea_hour = now.hour + 9
if korea_hour < 12:
  print("현재 오전 {}시입니다.".format(korea_hour))
if korea_hour >= 12:
  print("현재 오후 {}시입니다.".format(korea_hour))
```

16. 클래스

```
# 객체 생성
players = [
    {"name" : "정대만", "point3" : 24, "point2" : 6, "rebound":1},
    {"name" : "강백호", "point3" : 0, "point2" : 4, "rebound":10},
    {"name" : "서태웅", "point3" : 12, "point2" : 16, "rebound":5}
]
for player in players:
  score_sum = player["point3"] + player["point2"]
  print(player["name"], score_sum)

정대만 30
강백호 4
서태웅 28

# 함수로 객체 만들기
def create_player(name, point3, point2, rebound):
```

```
 return{"name":name,
        "point3":point3,
        "point2":point2,
        "rebound":rebound}
players = [
    create_player("정대만", 24, 6, 1),
    create_player("강백호", 0, 4, 10),
    create_player("서태웅", 12, 16, 5)
]
print(players)
```

```
[{'name': '정대만', 'point3': 24, 'point2': 6, 'rebound': 1}, {'name': '강백호', 'point3': 0, 'point2': 4, 'rebound': 10},
{'name': '서태웅', 'point3': 12, 'point2': 16, 'rebound': 5}]
```

```
def create_player(name, point3, point2, rebound):
 return{"name":name,
        "point3":point3,
        "point2":point2,
        "rebound":rebound}
def score_sum(player):
 return player["point3"] + player["point2"]
players = [
    create_player("정대만", 24, 6, 1),
    create_player("강백호", 0, 4, 10),
    create_player("서태웅", 12, 16, 5)
]
for player in players:
 print(player["name"], score_sum(player))
```

```
정대만 30
강백호 4
서태웅 28
```

```python
from os import name
# 클래스
class Player:
  # def __init__(self, 필요한 매개변수 값)
  def __init__(self, name, point3, point2, rebound):
    self.name = name
    self.point3 = point3
    self.point2 = point2
    self.rebound = rebound
players = [
    Player("정대만", 24, 6, 1)
    Player("강백호", 0, 4, 10)
    Player("서태웅", 12, 16, 5)
]
print(players[0].name)
print(players[0].point3)
print(players[0].point2)
print(players[0].rebound)
```

```
정대만
24
6
1
```

```python
# 클래스 안의 함수 = 메소드, 메서드 함수
from os import name


class Player:
  # def __init__(self, 필요한 매개변수 값)
  def __init__(self, name, point3, point2, rebound):
    self.name = name
    self.point3 = point3
    self.point2 = point2
    self.rebound = rebound
  def score_sum(self):
```

```
    return self.point3 + self.point2
players = [
    Player("정대만", 24, 6, 1),
    Player("강백호", 0, 4, 10),
    Player("서태웅", 12, 16, 5)
]

for player in players:
  print(player.name, player.score_sum())

정대만 30
강백호 4
서태웅 28
```

```
# 튜플 형태로 리턴하는 사용자 정의 함수 만들기
def test():
  return (1, 2)
a, b = test()
# a, b = (1, 2)와 같은 의미
print(a); print(b)   -> 1 ; 2
# 튜플 형태로 리턴하는 함수 복습
# 리스트와 인덱스와 값 반환 = enumerate()
for i, j in enumerate([1, 2, 3]):
  print("인덱스", i , "값", j)
인덱스 0 값 1   / 인덱스 1 값 2   / 인덱스 2 값 3
```

```
# 딕셔너리의 키와 값을 반환 = items()
for i, j in {0:1, 1:2, 2:3}.items():
  print("키", i, "값", i)
result
키 0 값 0
키 1 값 1
```

```
키 2 값 2
```