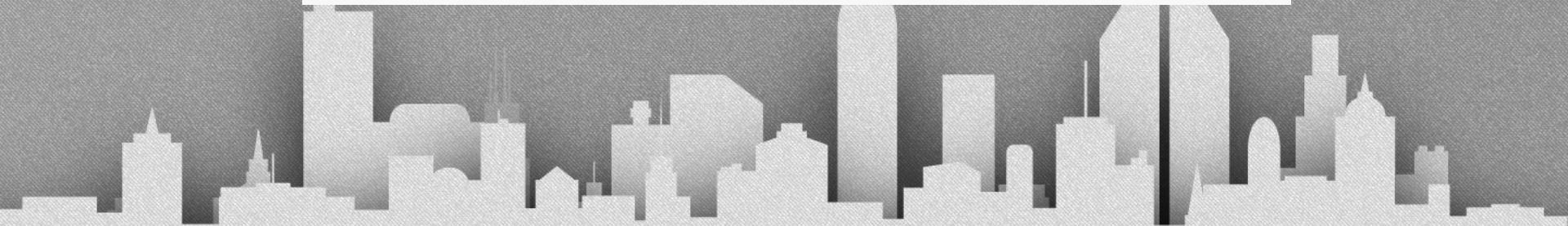


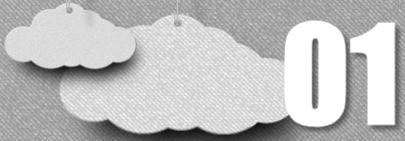
PYTHON

<https://www.youtube.com/c/EduAtoZPython>
Youtube 강의 채널 (채널 이름 : eduatoz 검색 가능)

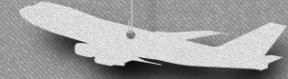


EduAtoZ - Python, Shell Script Study





01



파이썬 기본 type을 알아보자



what is python!?



- ▶ 파이썬은 간단하고 배우기 쉬운 오픈 소스 프로그래밍 언어이다
- ▶ 1989년 12월 귀도 반 로섬(Guido Van Rossum)이 개발한 인터프리터 언어이다



- ▶ 컴파일 과정이 없으며 플랫폼 독립적임
- ▶ 다른 언어로 작성된 모듈들을 연결하는 풀 언어 (Glue Language)로 이용됨
- ▶ 교육/실제 개발에서 많이 활용 됨
- ▶ python은 ‘귀도 반 로섬’이 즐겨 보던 영국 6인조 코미디 그룹 ‘몬티 파이썬’에서 따옴
- ▶ import this 를 입력하면 python의 철학을 볼 수 있음
- ▶ C로 구현되었으며, 다른 구현체와 구분해 언급할 때 CPython이라고 표기함
- ▶ 소스 코드 열람 <https://github.com/python/cpython>

파이썬의 특징



특징	설명
높은 가독성	<ul style="list-style-type: none">문법이 간결하여 가독성이 높음코드 블록 대신 들여쓰기 구분을 사용함들여쓰기에 탭을 사용하지 않을 것을 매우 강력하게 요구함
높은 생산성	<ul style="list-style-type: none">광범위한 라이브러리가 기본 제공됨확장성이 넓음, 개발자들이 수많은 모듈을 무료로 배포함
접착성	<ul style="list-style-type: none">기본 라이브러리 외에 쉽게 라이브러리 추가 가능C로 구현되어 있는 모듈을 쉽게 만들어 붙일 수 있음속도 문제가 있을 때 C로 구현하여 붙임
무한 정수	<ul style="list-style-type: none">메모리가 허용되는 범위에서 커다란 정수를 무한 사용 가능
동적 타이핑(typing)	<ul style="list-style-type: none">런타임시 타입 확인을 하고 자동 메모리 관리를 함
무료	<ul style="list-style-type: none">Python Software Foundation에서 관리Python Software Foundation License를 따름

Python Documentation



- 파이썬의 세부 사용법 설명은 Python Documentation을 참조한다
- <https://docs.python.org/3.7/index.html>

Python 3.7.6 documentation

Welcome! This is the documentation for Python 3.7.6.

Parts of the documentation:

What's new in Python 3.7?

or all "What's new" documents since 2.0

Tutorial

파이썬 학습

start here

built-in 함수 참고

Library Reference

keep this under your pillow

Language Reference

문법 학습

describes syntax and language elements

Python Setup and Usage

how to use Python on different platforms

Installing Python Modules

installing from the Python Package Index & other sources

Distributing Python Modules

publishing modules for installation by others

Extending and Embedding

tutorial for C/C++ programmers

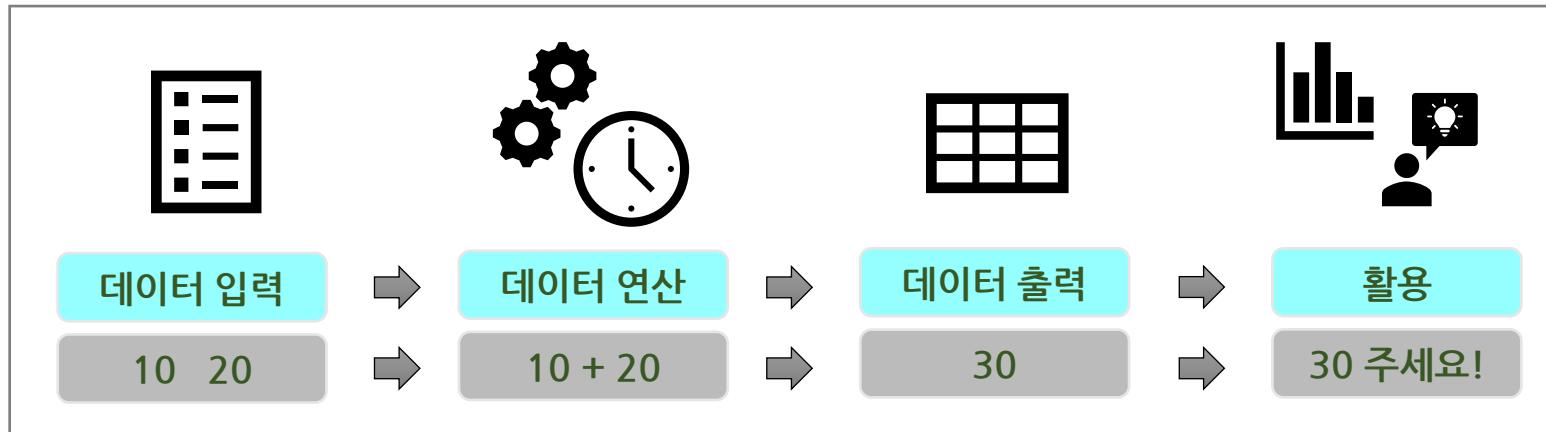
Python/C API

reference for C/C++ programmers

프로그램, 프로그래밍 언어



- ▶ 프로그램은 “정확한 결과가 요구되는 작업 처리를 위한” 명령 집합이다
- ▶ 명령의 대부분은 데이터를 읽고 연산한 뒤 그 결과를 출력하는 작업의 반복이다



- ▶ 프로그래밍 언어는 프로그래밍에 사용되는 규칙, 기호 등을 정의한 것이다
- ▶ 파이썬에서는 입력, 연산, 출력의 대상인 “데이터”가 ‘객체(object)’로 표현한다
 - 파이썬 학습의 대부분은 “객체의 종류와 그 객체의 특성 및 사용법”을 익히는 것이다
 - 먼저 프로그래밍 언어의 구성 요소를 살피고, 객체의 사용법을 익히도록 한다

Concept of class, instance, inheritance



```
class myInt:
```

```
    def __init__(self, value):
        super().__init__()
        self.real = value

    def __add__(self, other):
        return myInt(self.real + other.real)

    def __sub__(self, other):
        return myInt(self.real - other.real)

    def __str__(self):
        return "Go : " + str(self.real)
```

```
class yourInt(myInt):
    pass
```

class

instance

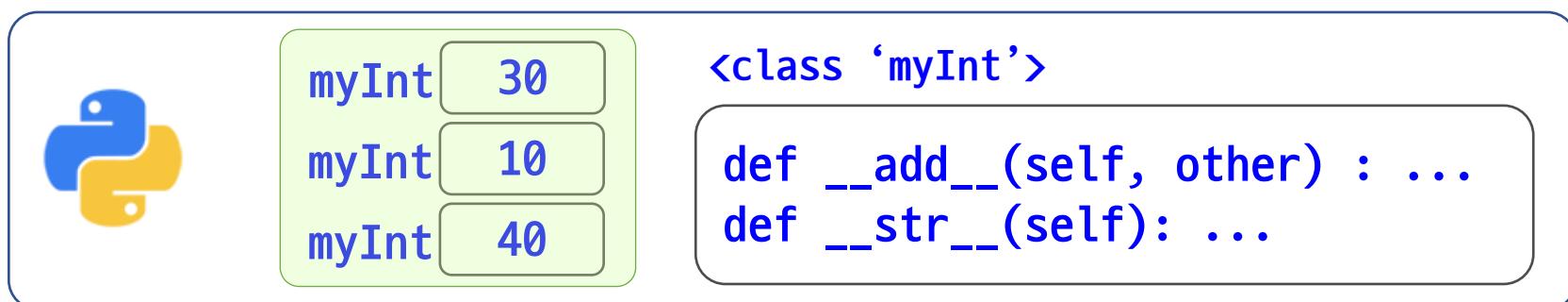
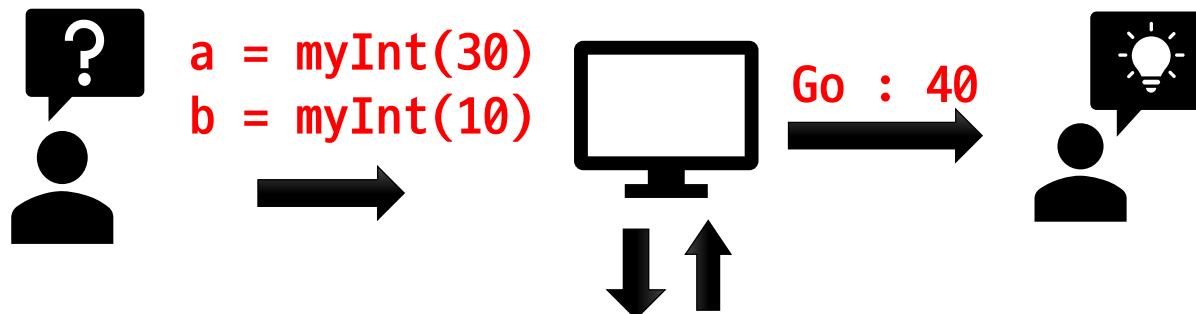
result

```
a = myInt(30)
b = myInt(10)
print(a)
print(b)
print(a + b)
print(a - b)

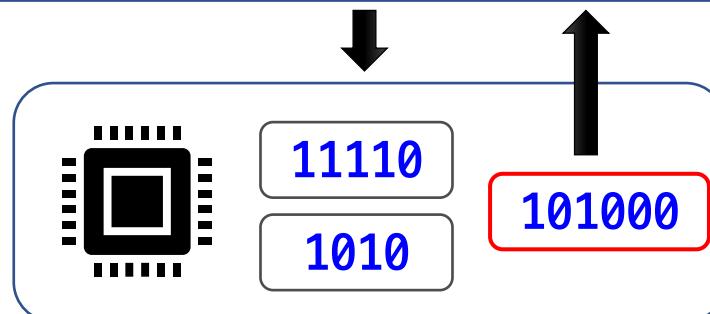
a = yourInt(30)
b = yourInt(10)
print(a)
print(b)
print(a + b)
print(a - b)
```

```
Go : 30
Go : 10
Go : 40
Go : 20
Go : 30
Go : 10
Go : 40
Go : 20
>>>
```

python에서의 연산



instance



Terms in Python - 1



object : “Everything is object in Python!!”

class와 instance를 모두 object로 통칭한다

class : type

attribute

method

attribute references

instantiation

instance : type object

attribute

method

attribute references

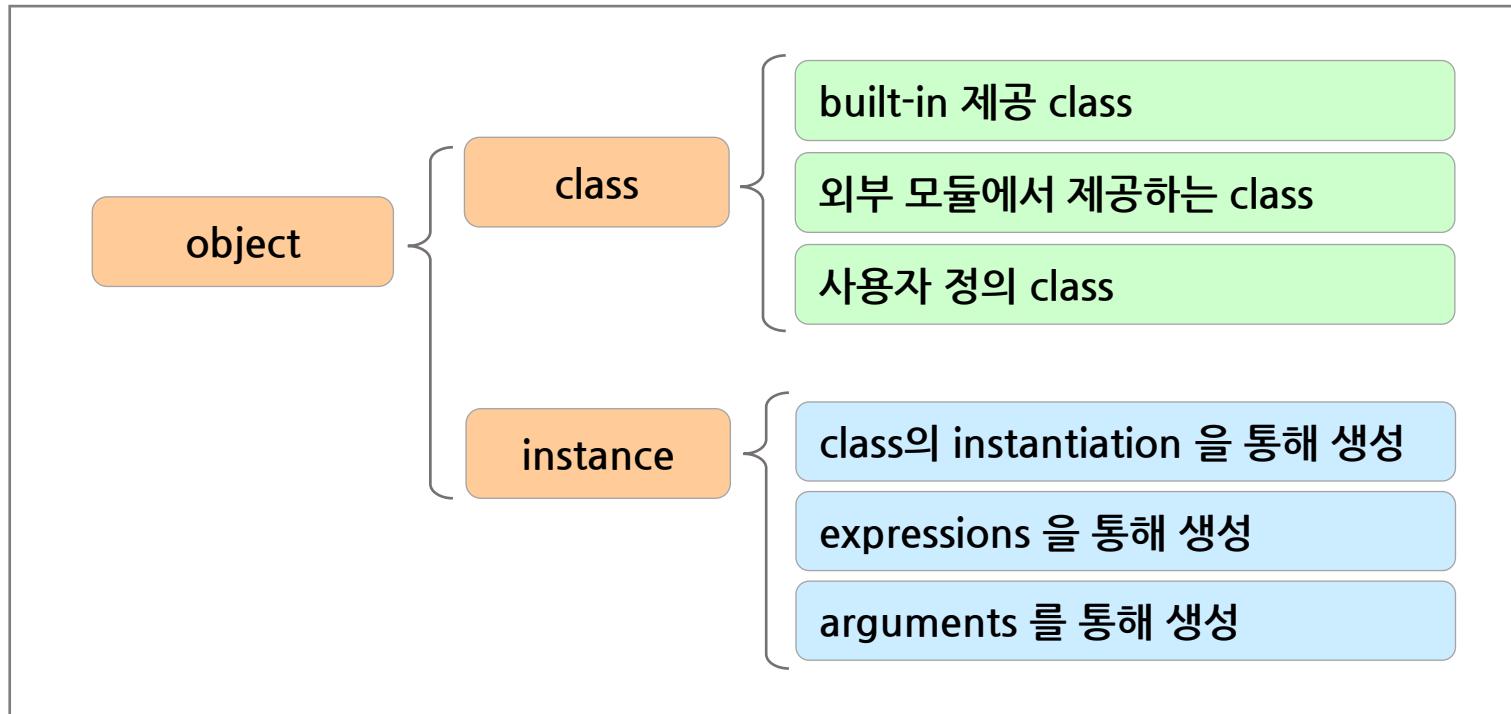
```
>>> map  
<class 'map'>
```

```
>>> a = map(int, (1, 2))  
>>> a  
<map object at 0x000002805E85D448>  
>>>
```

objects of Python



- 파이썬의 object 는 크게 class 와 instance 로 나눌 수 있다
- 대표적인 class의 종류와 instance 생성 방법은 다음과 같다



numeric type literals



decimal	binary	octal	hex
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F



int literals of python

decimal	13, -2, 1_3
binary	0b1101_0011
octal	0o15
hex	0xDE_A4, 0xab



float literals of python

소수점	3.14	-10.	.01	3.1_4
지수	0e0	-1e3	3.14e-2	

type : numeric



▣ Numeric types

type	설명
int	<ul style="list-style-type: none">▪ 양수, 0, 음수▪ 부호(+, -)를 사용할 수 있으며 숫자(0~9)로 표기한다▪ 10진 표기를 기본으로 사용한다▪ 0b, 0o, 0x 를 맨 앞에 붙여 2, 8, 16진수로 표기할 수 있다
bool	<ul style="list-style-type: none">▪ True, False▪ 연산 시 1, 0의 정수로 취급된다▪ 출력 시에는 "True", "False" 로 출력한다
float	<ul style="list-style-type: none">▪ 양수, 0.0, 음수▪ 부호(+,-)를 사용할 수 있으며 숫자(0~9)와 함께 소수점(.)을 사용한다▪ 소수점 표기법이 기본이며 하나이상의 숫자와 소수점(.)이 있어야 한다▪ 지수(exponential) 표기법을 사용할 수 있다 (예) 3.14e-2

▣ [퀴즈1] 아래의 수치 타입(numeric type)을 int, bool, float 로 구분해 봅시다

0xABCD True .1 99.99e1 123_456 .001 False 1 0.0 365 0b1100

type : str



👉 str 은 불변(Immutable) 값을 갖는 Container 의 하나이다

type	설명
str - 문자열	<ul style="list-style-type: none">▪ '...', ..." 과 같이 작은/큰 따옴표로 묶어 표시함▪ 문자열에 따옴표를 포함하기 위해 '와 "를 교차 사용가능▪ escape 문자(\)를 사용하여 따옴표 문자를 포함할 수 있음▪ 세 겹 따옴표('' or """)로 여러 줄에 걸친 문자열 표현가능▪ stringprefix 를 붙여 문자열의 종류 및 생성법을 표현함 → r, u, b, f 등이 있음 (raw, unicode, bytes, formatted)

String Literal

'hello' '12' "Good"

'name="JJ'" "name='JJ'"

'I\'m a girl' 'Good\nMorning'

'''Good
Morning'''

"""Good
Morning"""

불가능한 표현

'I'm a boy'

"my name is "Happy". "

\ 를 추가하여
가능한 표현으로 변경해 보세요.

type : containers



- Container는 불변(str, tuple)인 것과 가변(list, set, dict)인 것이 있다

type	example	설명
str	'hello'	<ul style="list-style-type: none">순서 있음, index를 사용하여 개별 요소 접근 가능index : 0부터 시작, 1씩 증가하는 정수, 위치 번호
tuple	(1, 2, 3, 4)	
range	range(1, 10, 2)	
list	[1, 2, 3, 4]	
set	{1, 2, 3, 4 }	<ul style="list-style-type: none">순서 없음, 개별 요소 접근 불가능집합 연산에 사용 (합집합, 교집합, 차집합 등)
dict	{'a' : 1, 'b' : 2 }	<ul style="list-style-type: none">순서 없음, key를 사용하여 개별 요소 접근 가능key : value와 match 되는 다양한 불변 타입

컨테이너 타입의 특징

- object의 identity 저장을 위한 여러 개의 공간(slot) 을 가지고 있음
- 개별 데이터를 item 또는 element 등으로 부름
- index 또는 key를 사용하여 개별 데이터에 접근 함 - subscription
- 매우 다양한 컨테이너 생성 방법이 있음(묶음 기호, comprehension, 생성자)

Object의 구성



Object는 Value, Type, Identity로 구성된다

type → int

value → 123 0x100 ← identity

float

0.5 0x200

type → list

value → 0x100 | 0x200 | 0x300 0x1000 ← identity

Immutable

100, 123 + 456 => 579

0.5, True, False, ‘abc’

(1, 2, 4)

Mutable

[1, 2, 3]

{10, 20, 30, 40}

{‘a’: 10, ‘b’: 20}

function for information of object



☞ 객체에 대한 정보를 알아 볼 수 있는 함수는 다음과 같다

built-in 함수	설명
type(객체)	<ul style="list-style-type: none">▪ 객체의 type을 반환 (객체.__class__ 와 동일)▪ 해당 객체를 instantiation 한 class는?
id(객체)	<ul style="list-style-type: none">▪ 객체의 identity를 반환 (객체가 저장된 메모리 주소)
isinstance(객체, 클래스)	<ul style="list-style-type: none">▪ 객체가 클래스의 instance 인지 확인▪ 객체가 클래스의 instance 이면 True 아니면 False 반환
dir(객체)	<ul style="list-style-type: none">▪ 객체가 사용할 수 있는 주요 속성의 name 목록 반환

```
>>> int  
<class 'int'>  
>>> type(1)  
<class 'int'>  
>>> isinstance(1,int)  
True
```

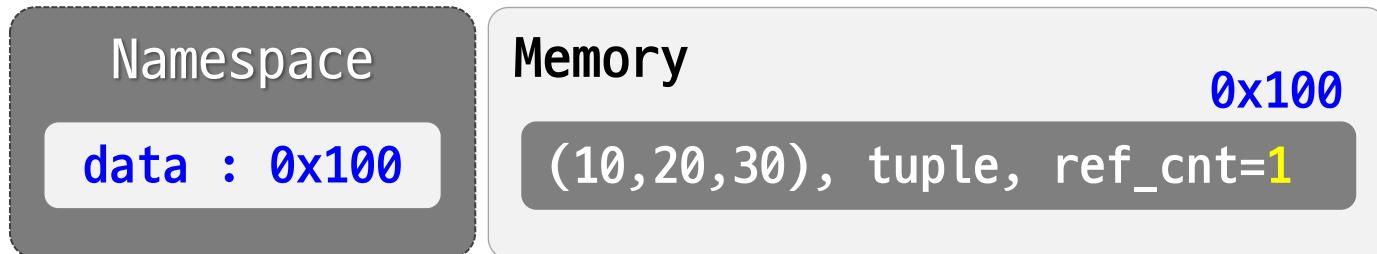
```
>>> str  
<class 'str'>  
>>> type('A+')  
<class 'str'>  
>>> isinstance('A+',str)  
True
```

assignment



여러 개 객체의 나열

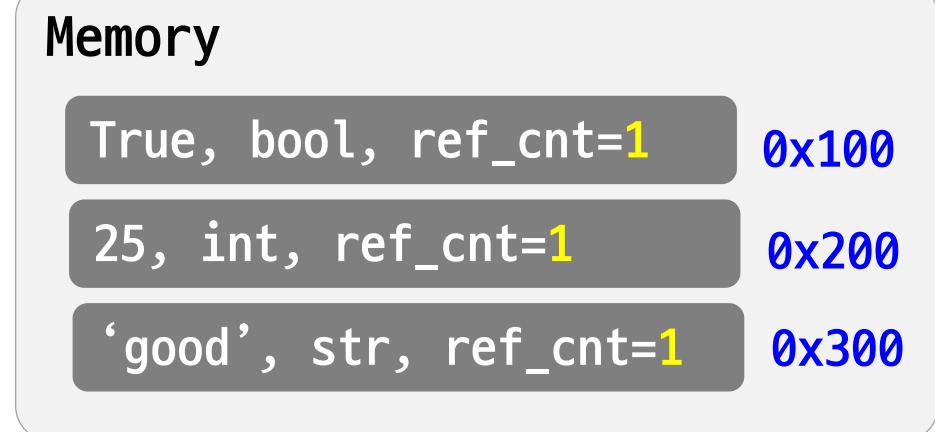
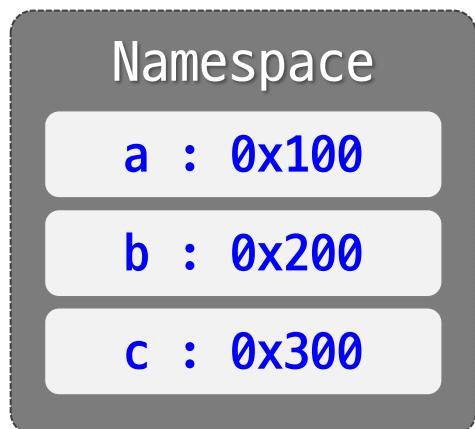
```
data = [10, 20, 30]  
       tuple
```



한 번에 여러 개의 변수(이름) 생성하기

```
a, b, c = True, 25, 'good'
```

unpack tuple



assignment / augmented assignment



assignment의 종류

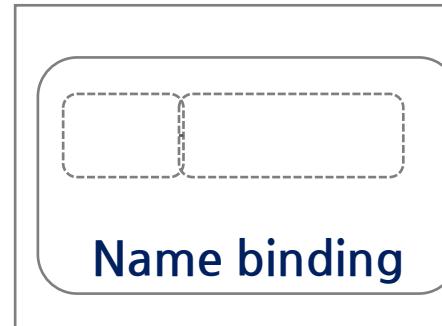
종류	기호	예	설명
assignment	=	a = 10	10에 a를 바인딩
augmented assignment	<code>**=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>//=</code> , <code>/=</code> , <code>%=</code> <code><<=</code> , <code>>>=</code> , <code>&=</code> , <code> =</code> , <code>^=</code> , <code>@=</code>	a += 10	a에 10을 더한 결과 객체에 a를 바인딩

assignment는 expression이 아닌 statement이다

- 프로그램 작성시 expression의 구분은 매우 중요하다
- Syntax에서 사용할 수 있는 위치가 다르기 때문이다 (함수, 제어문에서 다룬다)

```
>>> a = 1
>>> id(a)
1805149360
>>> a += 1
>>> id(a)
1805149376
```

다음의 결과는?
>>> b += 1
>>> type(b)



int instance object
real : 1 0x1000
int instance object
real : 2 0x2000

Name, Name binding



Object의 구분을 위해 이름(name, identifier, variable, reference) 필요

```
a = 2020 assignment  
b = 0.5  
print(a + b)
```

Memory

Object (2020, int, ref_cnt=1) 0x100

Object (0.5, float, ref_cnt=1) 0x200

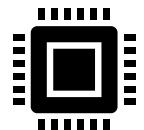


Namespace (→ dict type)

a : 0x100

b : 0x200

Name binding



0x100

Name binding 방법

assignment

import

class 정의

function 정의 등

a : 0x100 번지의 object를 reference 한다!

이름 작성 규칙



- 하나의 단어로 작성한다
- UNICODE 문자(한글포함), 숫자, _(under score)의 사용이 가능하다
- python 키워드(keyword)를 사용할 수 없다
- 숫자로 시작할 수 없으며, 영문자는 대/소문자를 엄격히 구분한다
- _로 시작하는 이름은 특별한 의미를 갖으므로 용도에 맞게 사용해야 한다

가능

value5
cnt
Cnt
DataFrame
합계
sum
__add__

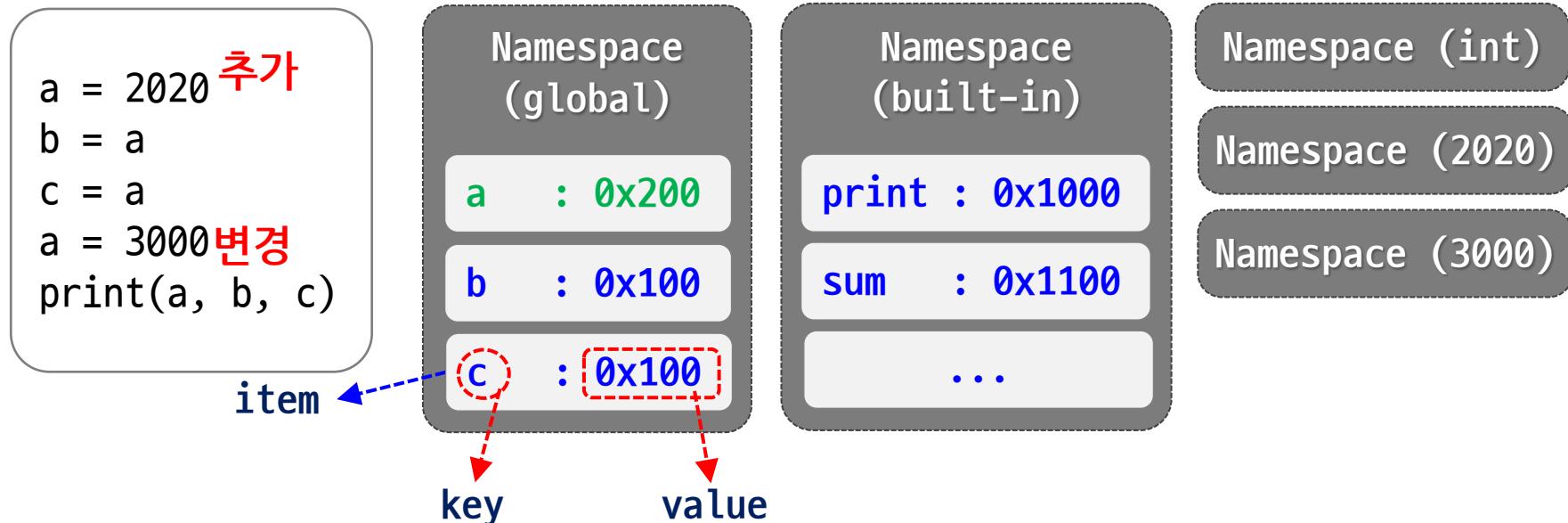
불가능

import
2020world
hello!
Milk Coffee
Cnt#2

Namespace의 이해



- _namespace는 이름 관리를 위한 **dict** container 이다
- 모든 **class, instance, function** 은 자신의 namespace를 갖는다



Memory

Object (2020, int, ref_cnt= 2)	0x100
Object (3000, int, ref_cnt= 1)	0x200

이름, Object의 제거



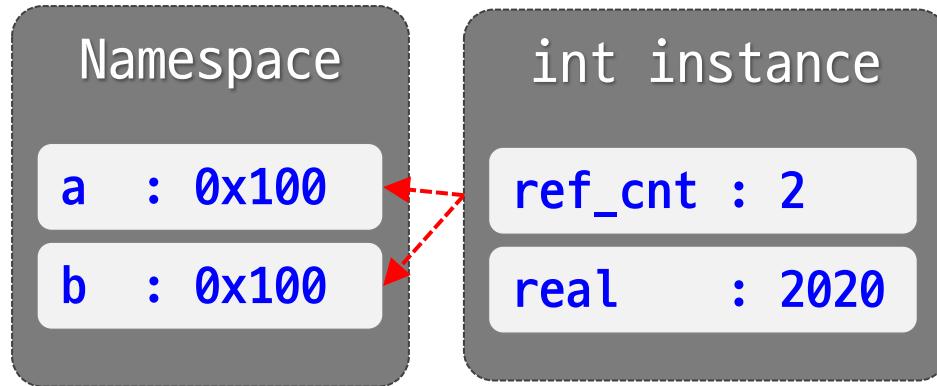
이름을 namespace에서 제거

del 이름

Object를 memory에서 제거

Garbage Collector가 ref_cnt == 0 인 것 제거

```
>>> a = 2020
>>> b = a
>>> import sys
>>> sys.getrefcount(b)
3
>>> del a
>>> sys.getrefcount(b)
2
>>>
```



0x100

이름의 특징



- 여러 개의 이름이 하나의 Object를 참조할 수 있다
- 이름은 참조하는 Object가 변경 될 수 있다 (ref_cnt : 자동 증가/감소)

```
>>> a = 2020 추가      >>> import sys  
>>> b = a              >>> sys.getrefcount(a)  
>>> c = a              2  
>>> a = 3000 변경      >>> sys.getrefcount(b)  
                           3  
>>> print(a, b, c)    ↑  
3000 2020 2020          object의 ref_cnt 반환
```

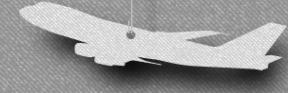
Namespace	
a	:0x200
b	:0x100
c	:0x100
sys	:0x300

Memory

Object (2020, int, ref_cnt= 3)	0x100
Object (3000, int, ref_cnt= 2)	0x200
Object (... , module, ref_cnt= ??)	0x300



02



파이썬 Programming 시작



Terms in Python - 2



package : module을 모아 놓은 폴더/디렉터리

module : python 프로그램 파일

function : 입력 데이터를 처리하여 결과를 반환하는 **명령**의 묶음

statement

- keyword, identifier, operator, delimiter 등으로 작성
- keyword : 의미를 가진 단어
- identifier : 구분을 위한 이름

class

method,
attribute

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

 return a + b

parameter : argument와 연결되는 이름

return : 호출한 쪽으로 반환하는 keyword

>>> add(**10, 20**)

30

함수 호출

argument : 함수에 전달하는 object

keyword



▣ 다음은 파이썬의 키워드 목록이다

▣ 키워드는 이름으로 사용할 수 없으며, 쉘에 다음을 입력하여 목록을 확인할 수 있다

```
>>> import keyword  
>>> print(keyword.kwlist)
```

Keywords				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

- 파이썬에서 기본 제공되는 함수인 built-in 함수는 이름으로 사용할 수 있다
- 그러나 혼란을 방지하기 위해 사용하지 않는 것을 권장한다

예) sum, min, max, int, str ... 등

function, method



▶ function과 method 는 function type의 object이다

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

```
>>> add(10, 20)  
30  
function call
```

```
class Num:  
    def __init__(self, v):  
        self.v = v  
    def __add__(self, other):  
        return self.v + other.v
```

```
>>> a = Num(10)  
>>> b = Num(20)  
>>> a.__add__(b)  
30  
>>> a + b  
30  
method call
```

```
>>> add  
<function add at 0x000002805E813558>
```

```
>>> Num.__add__  
<function Num.__add__ at 0x000002805E867288>
```

function : print



print 는 '문자열'을 출력하는 함수이다

```
print( *objects, sep=' ', end='\n', file=sys.stdout, flush=False )
```

*object

출력할 대상을 0개 이상 콤마(,)로 구분하여 전달함

sep=' '

출력 대상을 구분하는 구분자로 default argument는 공백

end='\n'

모든 대상을 출력 후 마지막 출력 내용, 기본값은 줄바꿈

file=sys.stdout

출력 위치, 기본값은 표준출력장치(콘솔)

print()

빈 줄 출력 (줄바꿈 동작)

print(obj1, obj2 ...)

print(str(obj1), str(obj2) ...)

print(a, b) → str(a), str(b)의 반환 값이 출력됨

문자열 생성



format 있는 문자열 생성 방법

format method

```
print('{ } { }'.format(a, b))
```

f-string

```
print(f'{a} {b}')
```

% operator

```
print('%d %d' % (a, b)) # %d : format 지시자
```

```
a = 10  
b = 20
```

```
print('[] + [] = {}'.format(a, b, a+b))  
print(f'{a} + {b} = {a+b}')  
print('%d + %d = %d' % (a, b, a+b))
```

```
10 + 20 = 30  
10 + 20 = 30  
10 + 20 = 30
```

format method



▶ name = 'Julie' age = 42 f1=11.3456, a=[6, 2, 4], b={'x':1, 'y':2} 인 경우

예시	설명	결과
'{}, {}'.format(name, age)	순서대로 {}에 대입, {} 사이 공백 없음	Julie, 42
'{} {}'.format(name)	중괄호, 중괄호로 감싼 값	{ } Julie
'{:.4}-{:.5.3}'.format(f1, f1)	{:.4} 4개 숫자 표시(반올림) {:.5.3} 5자리 확보, 3개 숫자 표시	11.35- 11.3
'{:>5}-{:>5}'.format(12,34)	{:>5} 오른쪽 맞춤, {:>5} 오른쪽 맞춤 후 0 채우기	12-00034
'{:<5}-{:0^5}'.format(12,34)	{:<5} 왼쪽 맞춤, {:0^5} 가운데 맞춤 후 0 채우기	12-03400
'{:,}'.format(123456789)	3자리마다 콤마 삽입	123,456,789
'{0}, {1}'.format(name, age)	position 번호 사용 (0번 부터)	Julie, 42
'{a} {b}'.format(b=10, a='X')	keyword로 객체 출력 위치 지정	X 10
'{0} {2} {1}'.format(*a)	argument unpack (sequence, set 타입)	6 4 2
'{x} {y}'.format(**b)	argument unpack (dict 타입)	1 2

f-string(Formatted string literals)



▶ name = 'Julie' age = 42 f1=11.3456 인 경우

예시	설명	결과
f'{name}, {age-20}'	{ } 내부에 expression 기입	Julie, 22
f'{f1:6.4}'	expression : 전체 자릿수.표시 숫자 개수 전체 자릿수에는 소수점 포함	▽ 11.35
f'name = {name}'	그대로 출력되는 문자열은 { } 외부에 기입	name = Julie
f'{} {{{name}}}'	중괄호, 중괄호로 감싼 값	{ } {Julie}
f'{age:>5}-{age:0>5}'	{expr:>숫자} 오른쪽 맞춤, {expr:0>숫자} 오른쪽 맞춤 후 0 채우기	▽ ▽ ▽ 42-00042
f'{age:<5}-{age:0^5}'	{:<숫자} 왼쪽 맞춤, {:0^숫자} 가운데 맞춤 후 0 채우기	42 ▽ ▽ ▽ -04200
f'{123456789:,}'	3자리마다 콤마 삽입	123,456,789

- https://docs.python.org/3/reference/lexical_analysis.html#f-strings
- 함수가 아니므로 argument unpack, argument position/keyword 사용 못함

function : input



▶ 키보드로 텍스트를 입력하고 Enter를 누를 때까지 기다려 입력을 받는다

input('메시지')	메시지를 출력하고 입력을 기다림
input()	메시지 출력 없이 입력을 기다림
입력 가능 문자	숫자, 문자열 모두 입력 가능, 공백이 포함되어도 상관없음
입력 진행	Enter를 눌러야 입력 진행이 됨
입력 데이터 타입	입력 데이터는 str로 반환 되므로 필요시 형 변환하여 사용

```
x = input('data : ') data : soyoung
print(x)                      soyoung

y = input()
print(type(y), y)            123
                            <class 'str'> 123
```

type 변환



- ▣ 다음의 built-in 함수들을 이용하여 형 변환을 할 수 있다
- ▣ 실제는 클래스의 생성자를 사용하여 객체를 만드는 작업이다

형식	기능	사용 예	결과
str(변환데이터)	문자열로 변환	str(10) str(3.14) str('apple')	'10' '3.14' 'apple'
int(변환데이터)	정수로 변환 base의 default는 10진수 (base=10)	int('13') int(4.89) int(-10) int('13', base=8)	13 4 -10 11
float(변환데이터)	실수로 변환	float(5) float('3.14') float(1.78)	5.0 3.14 1.78

8진수 13을
10진수로 변환

int('3.14') Error

float('ABC') Error

Line Structure



python 프로그램은 여러 개의 논리적, 물리적인 행들로 나뉜다

Logical Lines

Logical Line의 끝은 NEWLINE 토큰으로 표현됨
하나 이상의 Physical Line으로 구성 가능

Physical Lines

ENTER를 누르면 만들어지는 하나의 행

Comment

#으로 시작하며 Logical Line을 종료 시킴

명시적 행 결합

- ₩ 기호를 이용하여 하나의 Logical Line으로 결합
- 이어지는 줄의 들여쓰기도 중요함
- ₩ 는 주석을 결합하지 못하므로 ₩ 뒤에 주석 사용할 수 없음

묵시적 행 결합

- (), [], { } 가 사용 되는 표현은 ₩ 없이도 결합
- 이어지는 줄의 들여쓰기는 중요하지 않음
- 묵시적 행 결합하는 행 사이에는 NEWLINE 토큰 생성되지 않음
- 주석을 포함할 수 있음

statement



statement는 실행가능한(executable) 최소의 독립적인 코드이다

simple statement

한 개의 Logical Line으로 구성, ";"으로 여러 개 나열 가능

compound
statement

다른 statements를 포함

다른 statements의 실행을 제어하거나 영향을 줌

simple statement

expression statements

assignment

del

global

nonlocal

assert

yield

raise

import

continue

break

return

pass

compound statement

if

조건에 따른 분기를 위한 구문

while

조건에 따른 반복을 위한 구문

for

지정된 횟수 만큼 반복을 위한 구문

try

예외 처리를 위한 구문

with

리소스 핸들 자동 반환을 위한 구문

def

함수 정의

class

클래스 정의

식(expression)



expression은 한 개의 객체로 evaluate 될 수 있는 것을 의미한다

Arithmetic conversions

$1+3.14 \rightarrow \text{float}$, $1+3j \rightarrow \text{complex}$, $'%d' \% 10 \rightarrow \text{str}$

Atoms

identifiers(names)

literals

enclosures

Primaries

attribute references

subscriptions

slicings

Operations

await expression

power

shift

calls

unary bitwise

binary bitwise

unary arithmetic

binary arithmetic

comparisons

boolean operations

Conditional expressions

조건 결과의 참/거짓에 따라 다른 객체

Lambdas

람다 표현식은 이름 없는 함수 객체

Expression lists

콤마(,)로 구분된 expression의 나열 \rightarrow tuple 객체

Operators & Delimiters



Operator와 Delimiters 목록

Operators							
+	-	*	**	/	//	%	@
<<	>>	&		^	~	matrix multiplication	
<	>	<=	>=	==	!=		
Delimiters							
()	[]	{	}	assignment	
,	:	.	;	@	=	annotation	
+ =	- =	* =	/ =	// =	% =	@ =	
& =	=	^ =	>> =	<< =	** =	augmented assignment	

- Operators 와 Delimiters 중 () [] { }, . 는 expression에서 사용된다
- assignment, augmented assignment는 expression이 아닌 statement이다

연산 종류 및 우선 순위



순위	종류	연산	설명
1	괄호 묶기, 디스플레이	(expressions...), [expressions...], {key:value...}, {expressions...}	parenthesized expression {list, dictionary, set} display
2	Primaries	x[index], x[index:index], x(args...), x.attribute	Subscription, Slicing, Function Call, Attribute reference
3	거듭제곱	**	Exponentiation
4	단항	+x, -x, ~x	Positive, Negative, Bitwise not
5	산술	*, /, //, %, @	Multiplication, Division, Remainder
6	산술	+, -	Addition, Subtraction
7	비트 시프트	<<, >>	Bitwise shifts
8	비트	&	Bitwise AND
9	비트	^	Bitwise XOR
10	비트		Bitwise OR
11	관계	in, not in, is, is not, <. <=, >, >=, <>, !=, ==	Comparisons, {membership, identity} test
12	논리	not x	Boolean NOT
13	논리	and	Boolean AND
14	논리	or	Boolean OR
15	조건표현식	x if C else y	Ternary operator
16	람다표현식	lambda	Lambda expression
17	대입식	:=	Assignment expression (version 3.8)

거듭제곱과 산술 연산



연산은 우선 순위에 따라 진행된다

우선순위 표는 필요시 참조용으로 사용, 거듭제곱과 산술 연산 우선 순위는 암기한다

1	거듭제곱	**	$2 ** 4 \rightarrow 16$
	나누기(몫)	//	$10 // 3 \rightarrow 3, 10.5 // 1.2 \rightarrow 8.0$
2	나머지	%	$10 \% 3 \rightarrow 1, 10.5 \% 1.2 \rightarrow 0.9000000000000004$
	나누기	/	$10 / 3 \rightarrow 3.333333333333335$
	곱하기	*	$2 * 5 \rightarrow 10$
3	빼기, 더하기	- , +	$5 - 2 \rightarrow 3, 3 + 4 \rightarrow 7$

우선 순위가 같은 경우 왼쪽 → 오른쪽으로 연산이 이루어짐 $2 + 3 - 2 * 4 = ?$

() 괄호를 이용하여 우선 순위 변경할 수 있음 $(2 + 3) * 4 = ?$

%의 피연산자는 ‘양수’어야 함 (양의 실수 가능)

비교 연산(Comparison)



▶ 비교 연산의 결과는 True 또는 False 이다

비교 연산	기능
is [not]	객체의 identity(id()결과) 비교
[not] in	멤버십 검사 연산
==	같다
!=	다르다

비교 연산	기능
<	작다
>	크다
<=	작거나 같다
>=	크거나 같다

==, !=

모든 값에 대한 비교, 자료형에 따라 다른 비교 방법이 적용됨

[not] in

여러 번 반복하여 사용할 경우 프로그램 성능이 낮아짐

a < b < c

비교 연산을 연결해서 사용 가능함

논리 연산(Boolean operations)



복잡한 조건의 표현이 필요할 때 사용하며 and, or, not 연산이 있다

and, or : 연산자의 앞 또는 뒤의 객체를 반환

not : True 또는 False를 반환

False

False, None, 모든 형의 숫자 0, 빈 문자열과 컨테이너

Short circuit

x and y : x 값을 구해 거짓인 경우 x를 반환, 참이면 y 값을 구해 결과 반환

x or y : x 값을 구해 참인 경우 x를 반환, 거짓이면 y 값을 구해 결과 반환

and	결과
True and True	True
True and False	False
False and True	False
False and False	False

or	결과
True or True	True
True or False	True
False or True	True
False or False	False

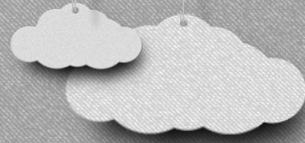
not	결과
not True	False
not False	True

산술 및 비교 연산을 활용한 조건 작성 실습



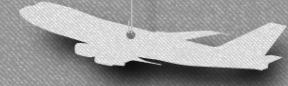
- ▣ 다음의 조건을 작성하여 보도록 한다
- ▣ 여러 방법이 있는 경우 여러 가지를 적어도 좋다

a와 b 가 같은 값을 갖는다	
a가 b 이상이다	
a가 짝수 이다	
a가 3의 배수이다	
a는 80보다 크고 90보다 작다	
a와 b는 같은 객체이다	
a는 정수이고 b는 문자열이다	
a는 100보다 크거나 0보다 작다	



03

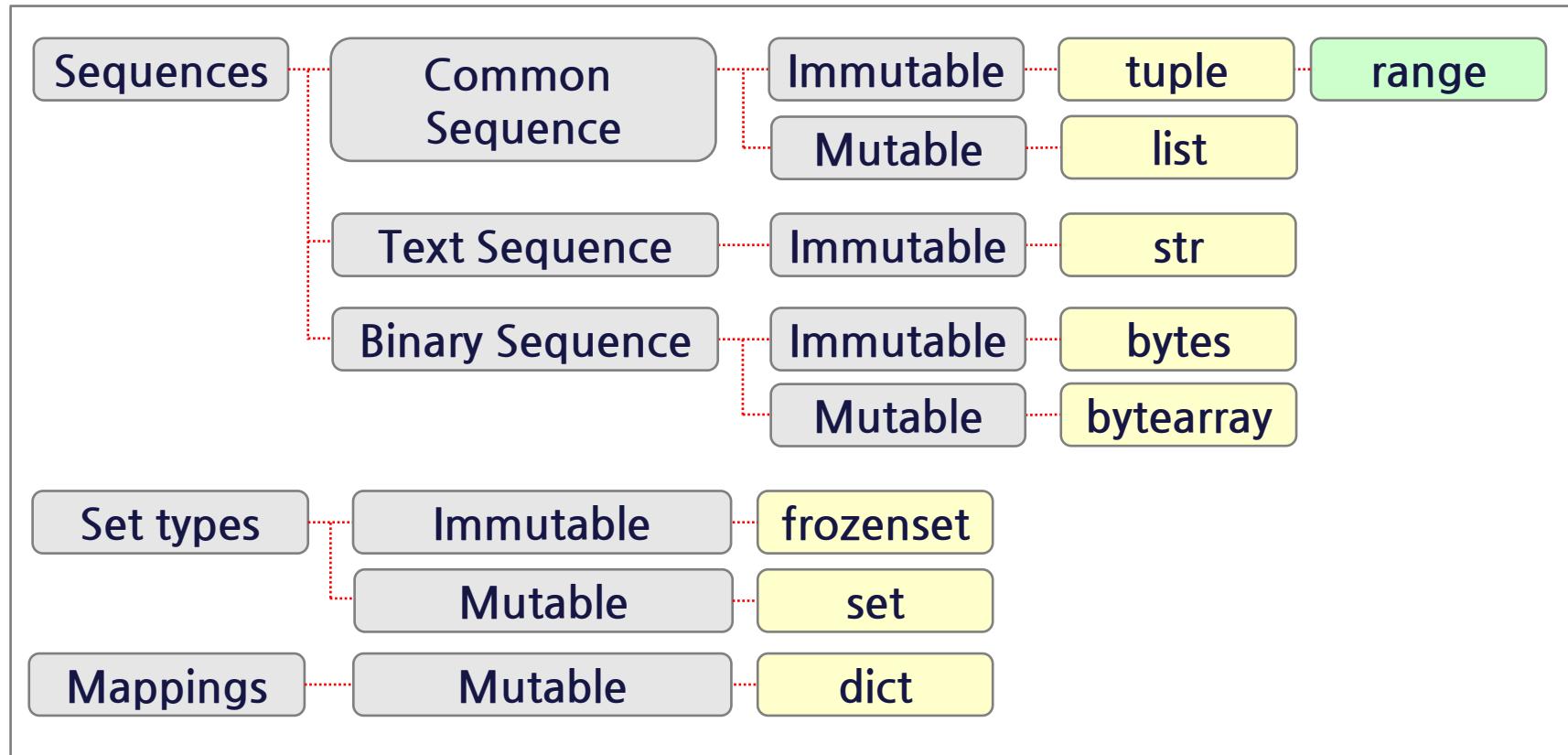
Container의 생성 및 methods



컨테이너(Container)의 분류



Container는 여러 item을 가질 수 있는 type이다



컨테이너에는 객체를 저장할 수 있는 슬롯(slot, 여러 개 공간)이 있음(자동 증가)

묶음기호를 사용한 Container 생성



- 임시 데이터, 참조용 데이터를 포함한 또는 empty Container 생성
- 묶음기호 사이에 item을 콤마(,)로 구분하여 나열하는 방법

1차원 구조

```
con = [item0, item1, item2, item3, item4] → 묶음 기호
```

2차원 구조

```
con = [(item00, item01, item02), (item10, item11, item12)]
```

```
con1 = [1, 2, 3, 4, 5]
```

```
con2 = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
con3 = {  
    'name': 'Julie',  
    'age': 20,  
    'phone': '010-123-1234'}
```

```
con2 = [(1, 2, 3),  
        (4, 5, 6),  
        (7, 8, 9)]
```

Better!

(), { }, [] → 묶시적 줄바꿈

‘ ’ “ ” → \를 사용한 명시적 줄바꿈

Container별 묶음 기호 및 사용 가능 item type



tuple

()

제한 없음

묶음 기호 없이 나열 가능, item이 한 개인 경우
뒤에 콤마(,) 사용해야 함 (예) a = 1,

list

[]

str

“ ”

“ ”

문자(Unicode)

콤마 사용 없이 나열

set

{ }

hashable 객체, { } 는 empty dict 객체임

dict

{ k : v }

key → hashable 객체 value → 제한 없음

hashable → hash()에 의해 값을 반환 받을 수 있는 객체
예) int, float, tuple, str 등 (immutable 특성의 객체)

```
>>> a = (1, 2, 3)  
>>> hash(a)  
2528502973977326415
```

hash 함수



• **hash(object)**

→ integer

object의 저장된 내용을 기준으로 한 개 정수를 생성하여 반환

빠른 비교를 위해 사용

hash 가 같으면 같은 객체 취급

많은 데이터도 한 번에 비교 가능

2528502973977326415

hash

(1, 2, 3)

hashable

TypeError

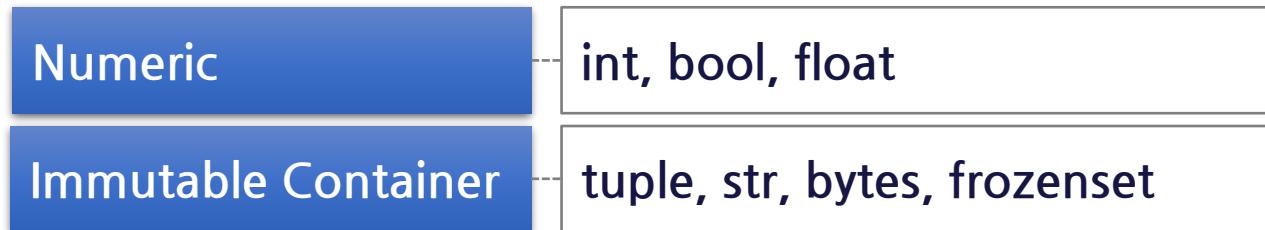
hash

[1, 2, 3]

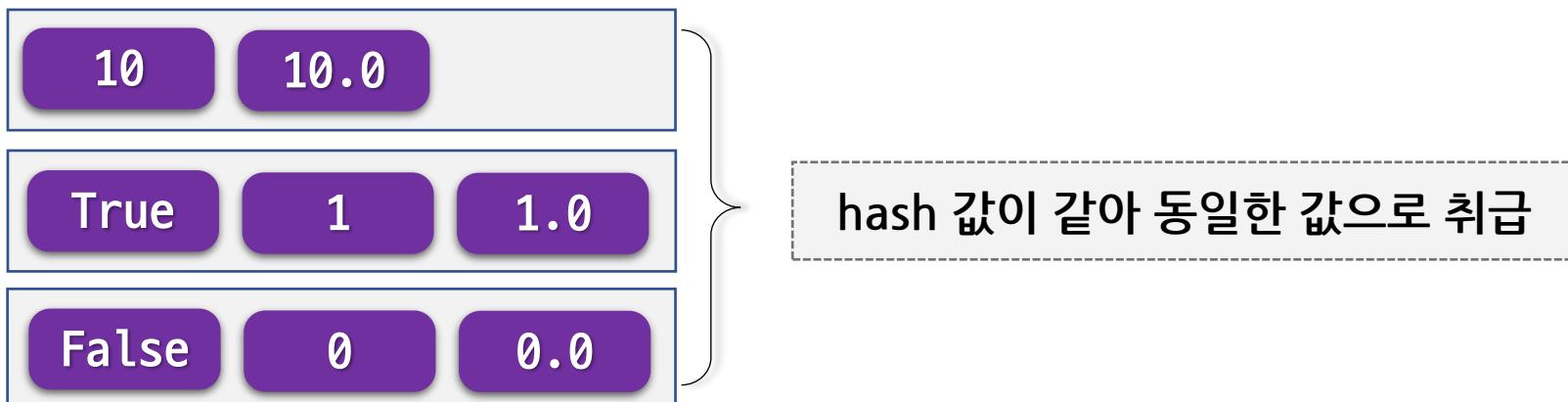
Hashable



- ‘hash()’ 함수로 값을 받아 올 수 있으면 hashable이다
- Numeric 과 Immutable Container는 hashable 이다



- Numeric 값들은 타입이 다르더라도 hash로 비교하여 동등비교를 한다



“set의 item, dict의 key는 빠른 중복 판단을 위해 hashable 객체를 사용함”

생성자를 사용한 Container 생성



▶ 생성자를 사용한 Container 생성은 주로 type 변경에 사용된다

▶ 생성자는 type과 이름이 같은 함수로, type의 instance를 생성하여 반환한다

tuple()

tuple(iterable)

- tuple() 또는 () 은 empty tuple 생성
- 생성 후 item 추가, 간신, 제거 불가능

list()

list(iterable)

- list() 또는 [] 은 empty list 생성
- 생성 후 item 추가, 간신, 제거 가능

str()

str(object)

- str() 또는 “ ” 은 empty str 생성
- 문자열 표현을 제공하는 객체를 argument로 사용
- 생성 후 item 추가, 간신, 제거 불가능

set()

set(iterable)

- set() 은 empty set 생성
- 생성 후 item 추가, 제거 가능

dict()

dict(**pVK)

dict(mapping, **pVK)

dict(iterable, **pVK)

- dict() 또는 {} 은 empty dict 생성
- 생성 후 item 추가, 간신, 제거 가능

Comprehension을 사용한 Container 생성



list, set, dict 및 Generator를 생성하는 expression이다

List Comprehension	[]	작성방법은 동일하며 서로 다른 묶음 기호를 사용함
Set Comprehension	{ }	Generator는 Iterator처럼 next()로 다음 항을 받아 옴
Dict Comprehension	{ : }	GE는 argument로 사용시 묶음 기호 없이 사용 가능
Generator Expression	()	List는 python 2부터 제공되며 나머지는 3부터 제공됨



[x * 2 for x in range(5)]

[0, 2, 4, 6, 8]

[x * 2 for x in range(10) if x % 2]

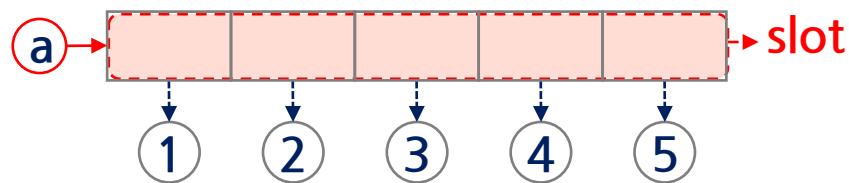
[2, 6, 10, 14, 18]

컨테이너 구조

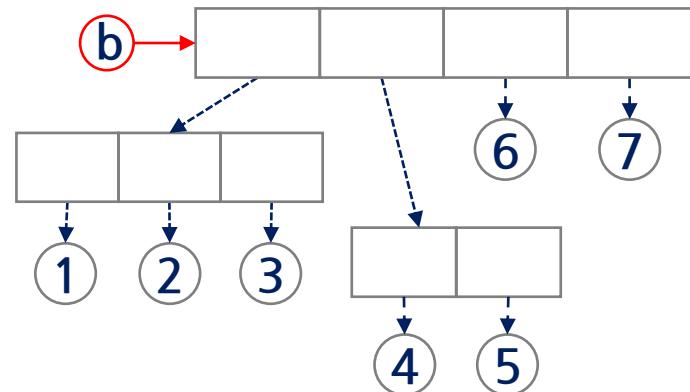


☞ Container는 객체의 identity를 저장하는 공간(slot)을 가진 객체이다

a = [1, 2, 3, 4, 5]



b = [[1, 2, 3], [4, 5], 6, 7]



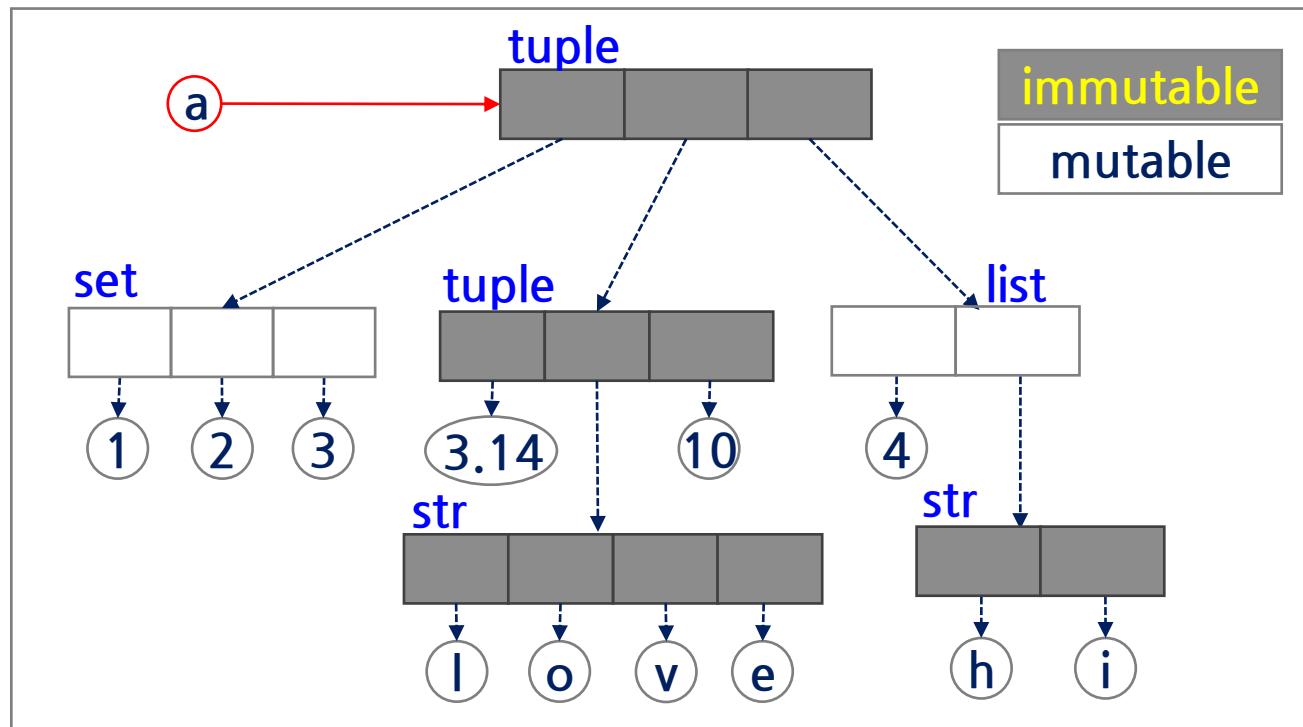
☞ Mutable Container의 slot 관리

- Mutable Container의 slot에 기록된 내용을 변경, 삭제, 추가 할 수 있다
- 저장 객체가 증가하면 slot의 크기가 자동 증가한다 (1개씩 증가 아님)
- 저장된 객체를 삭제하면 slot의 크기는 자동 감소(list) 또는 유지(set, dict)된다
- clear() 메서드를 사용하여 전체 삭제하면 slot이 모두 반납 된다

컨테이너 구조



```
a = ( {1, 2, 3}, (3.14, 'love', 10), [4, 'hi'] )
```



Sequence Type Container - 공통 연산



▶ Sequence Type의 Container에는 tuple, list, str, range 있다

◀ s, t : 동일 타입의 sequence container | n, start, stop, step : 정수 | x : 임의 객체

포함 여부 확인

x in s

'a' in 'apple'

True

x not in s

'a' not in 'apple'

False

일부 내용 얻기

s[n]

'apple'[0]

'a'

s[start:stop:step]

'watermelon)[:5]

'water'

slice

“range는 +, * 연산 불가능”

합치기

s + t

'in' + 'door'

'indoor'

반복하기

s * n (n * s)

'-'*5

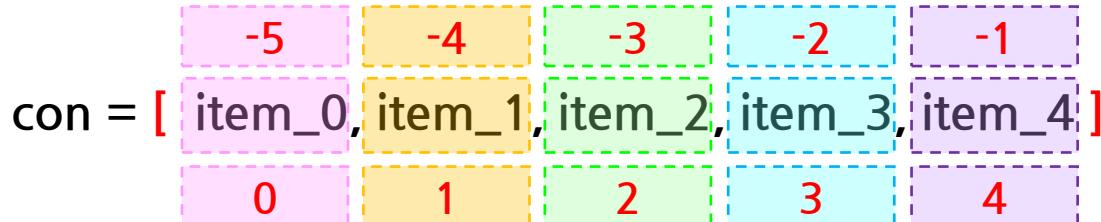
'-----'

“새로운 container 생성”

Sequence type Container의 item 접근



- tuple, list, str, range의 item은 순서 번호가 있으며, 이것을 Index라 한다



index 범위를 벗어나면
→ IndexError 발생

- Subscription은 특정 위치의 item을 가져오는 연산이다

s[n] index가 n인 item

s[n][m] index가 n인 item의 index가 m인 item

con2 = [(1, 2, 3),
(4, 5, 6),
(7, 8, 9)]

con2[0]

(1, 2, 3)

con2[1][-1]

6

con2[3]

IndexError: list index out of range

con3 = { 'name' : 'Julie',
'age' : 20,
'phone': '010-123-1234' }

con3['age']

20

con3['D']

KeyError: 'D'

slicing



- 👉 slicing은 Container의 일부를 동일 타입의 새로운 Container를 만든다

`s[start:stop:step]`

slice

start~stop-1 까지 step 만큼씩 건너 위치한 item들로 구성된 container 생성

`con1 = [1, 2, 3, 4, 5]`

`con1[1:3]`

[2, 3]

`con1[0:5:2]`

[1, 3, 5]

- 👉 start, stop, step에 해당하는 정수는 일부 또는 모두 생략할 수 있다

👉 생략 시 `start = 0, stop = len(s), step = 1`로 동작한다

👉 단, 모든 정수 생략 시 하나의 콜론(:)은 포함해야 한다

`s[:]`

모든 item

`s[::-step]`

0부터 마지막까지 step 만큼씩 건너뛴 item

`s[start:]`

start 부터 마지막까지의 item

`s[:stop]`

0 부터 stop-1까지의 item

`s[::-1]`

0부터 마지막까지 뒤집은 item

Mutable Sequence Type - 연산



list에서만 사용되는 연산으로 slot의 내용 및 길이 변경이 일어날 수 있다

s, t : list container | n, start, stop, step : 정수 | x : 임의 객체

“새로운 container 생성 아님”

numbers = [10, 20, 30, 40], temp = [1, 2]

갱신
(교체)

s[n] = x
s[start:stop:step] = t
s[:] = t (길이 무관)

numbers[0] = 100

[100, 20, 30, 40]

numbers[:2]=[1, 2]

[1, 2, 30, 40]

제거

del s[m]
del s[start:stop:step]

del numbers[2]

[10, 20, 40]

del numbers[:2]

[30, 40]

확장

s += t
s *= n

numbers += temp

[10, 20, 30, 40, 1, 2]

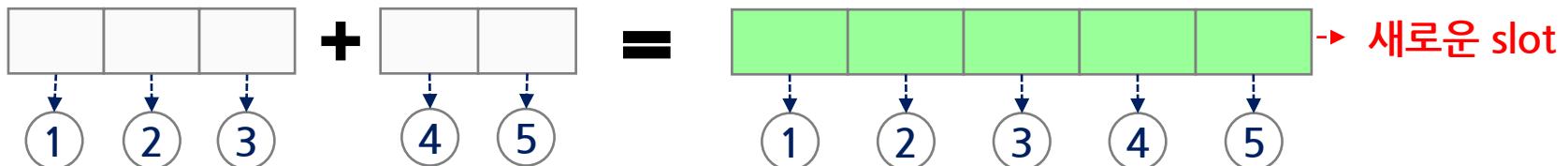
temp *= 3

[1, 2, 1, 2, 1, 2]

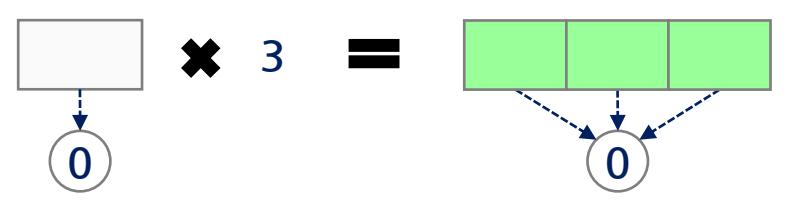
Container의 +, * 연산



[1, 2, 3] + [4, 5] → [1, 2, 3, 4, 5]

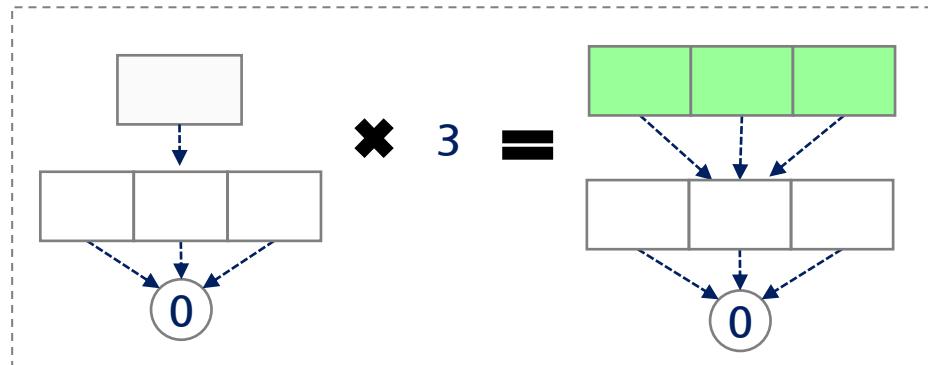
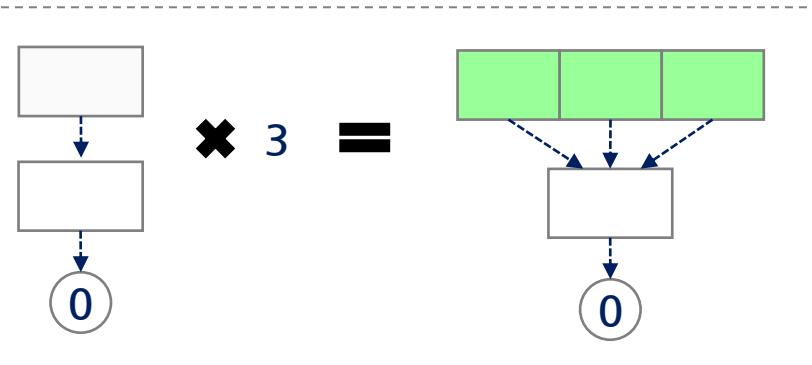


[0]*3 → [0,0,0]



[[0]] * 3 → [[0],[0],[0]]

[[0]*3]*3 → [[0,0,0],[0,0,0],[0,0,0]]

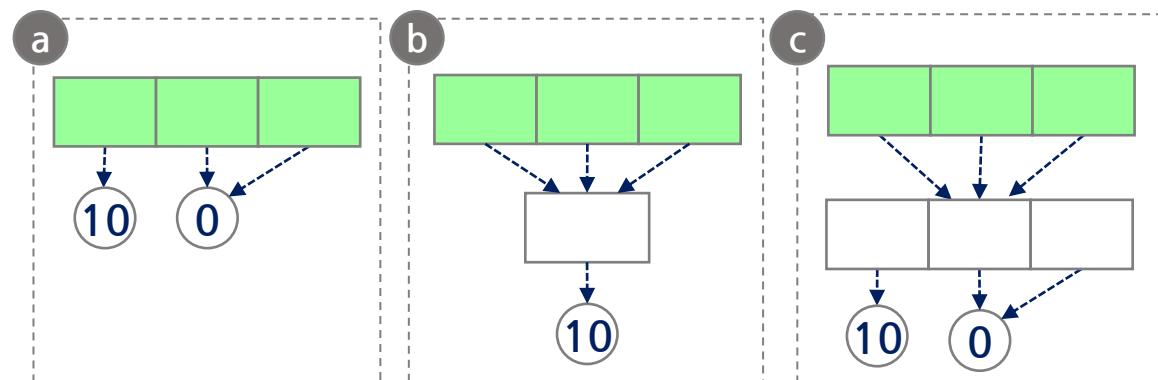
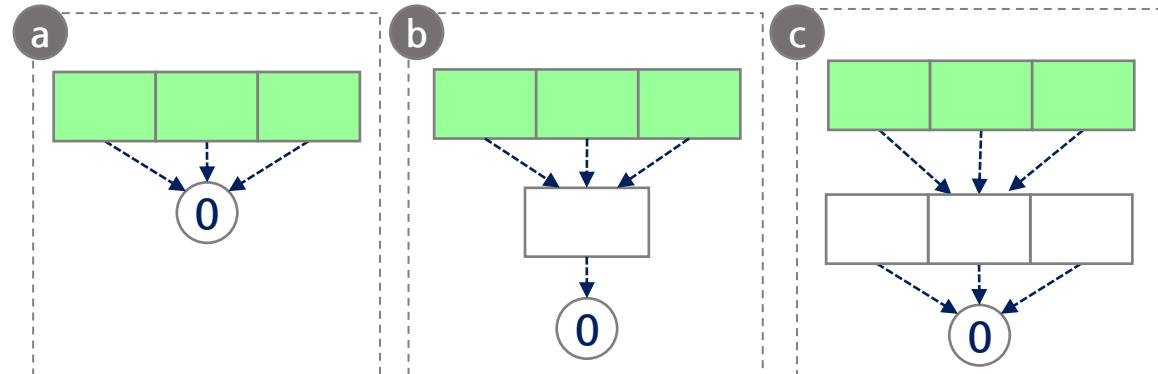


Container의 * 연산 주의 사항



▣ 다음 코드의 실행 결과는 무엇이며, 이유는 무엇인지 분석하라

```
a = [0] * 3  
b = [[0]] * 3  
c = [[0]*3] * 3  
  
a[0] = 10  
b[0][0] = 10  
c[0][0] = 10
```



Sequence Type - 공통 method



▶ Sequence Type Container에는 tuple, list, str, range 가 있다

x = object.method(arguments)

특정 값을 갖는 item의 index 구하기

s.index(x, [, start[, stop]])

특정 item의 개수

s.count(x)

fruits = ('apple', 'kiwi', 'apple', 'apple')

fruits.index('apple')

0

fruits.index('apple', 1)

2

fruits.count('apple')

3

fruits.index('apple', 1, 2)

ValueError: tuple.index(x): x not in tuple

튜플(tuple)의 구조 및 특징



▶ 튜플은 Common Sequence로 분류되며 **Immutable** 이다

생성 방법

(a, b, c) tuple(iterable) x = a, (a,) x = a, b, c () tuple()

아이템

제한 없음

특징

Sequence Type의 공통 연산 사용([not] in, +, *, subscription, slicing)

hashable 이므로 set의 item, dict의 key로 사용 가능

동등비교(==)연산은 모든 아이템이 순서까지 같을 때 참(True)

튜플의 아이템 변경(assignment) 및 삭제(deletion) 불가능

불변의 특징 때문에 주로 참조 표(lookup table) 작성에 사용함

s.index(), s.count() 메서드를 사용할 수 있음

리스트(list)의 구조 및 특징



☞ 리스트는 Common Sequence로 분류되며 **Mutable** 이다

생성 방법	[] [a] [a, b, c] list() list(iterable) [x for x in iterable]
아이템	제한 없음
특징	Sequence Type의 공통 연산 사용([not] in, +, *, subscription, slicing) 동등비교(==)연산은 모든 아이템이 순서까지 같을 때 참(True) mutable sequence type의 연산 및 메서드 사용 (=, del, +=, *=) item 변경, 삽입, 삭제가 필요한 경우 사용함 item 삽입, 삭제, 정렬을 위한 다양한 method가 제공됨

리스트(list) - 삽입/추가 메서드



✓ s : list container, t : iterable, m : index 번호, x : 임의 객체

맨 뒤에 한 개 item 추가

s.append(x)

맨 뒤에 여러 item 추가

s.extend(t)

연산자 += 과 동일함

특정 위치에 item 삽입

s.insert(m, x)

Index 범위를 벗어나면 append 처리됨

s



list

t



iterable

s.append(30)



s.extend(t)



s.insert(1, 15)



s.insert(5, 15)



리스트(list) - 삭제 메서드



✓ s : list container, t : iterable, m : index 번호, x : 임의 객체

특정 위치의 item
반환하고 삭제

R=s.pop($m=-1$)

1개 item 반환, m 생략 시 마지막 item 반환

IndexError : index의 사용 범위를 벗어날 때

특정 item 삭제

s.remove(x)

동일 값을 갖는 item 중 맨 처음 1개 삭제

모든 item 삭제

s.clear()

ValueError : 동일 값을 갖는 item이 없을 때

s

10 20 30 40 30

list

R = s.pop(2)

10 20 40 30

R

30

s.remove(20)

10 30 40 30

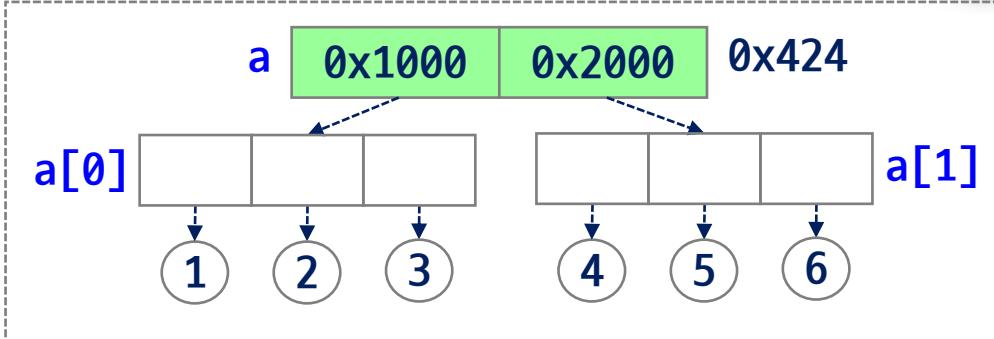
s.clear()



copy, deepcopy

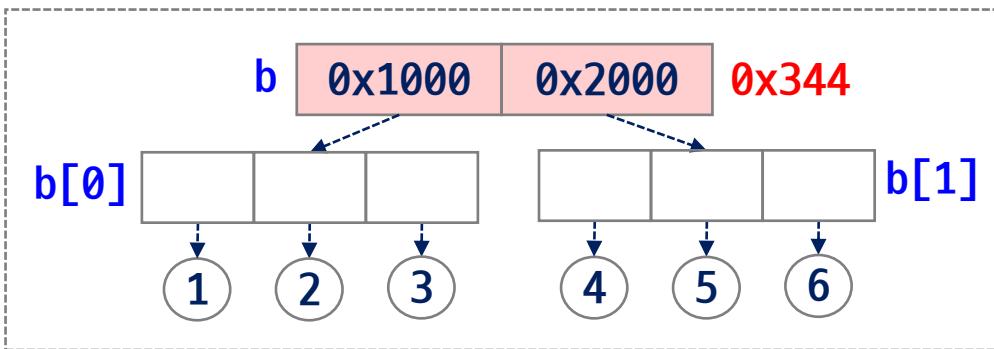


```
a = [ [1,2,3], [4,5,6] ]
```



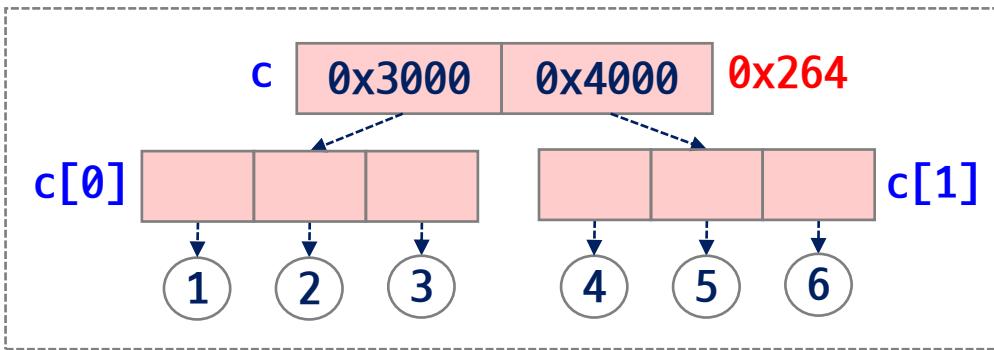
```
b = a.copy() # shallowcopy
```

a와 같은 크기의 slot을
새로 생성하고, 내용 복사



```
import copy  
c = copy.deepcopy(a)
```

a, a 내부에 포함된 container 모두!
slot 새로 생성 및 내용 복사



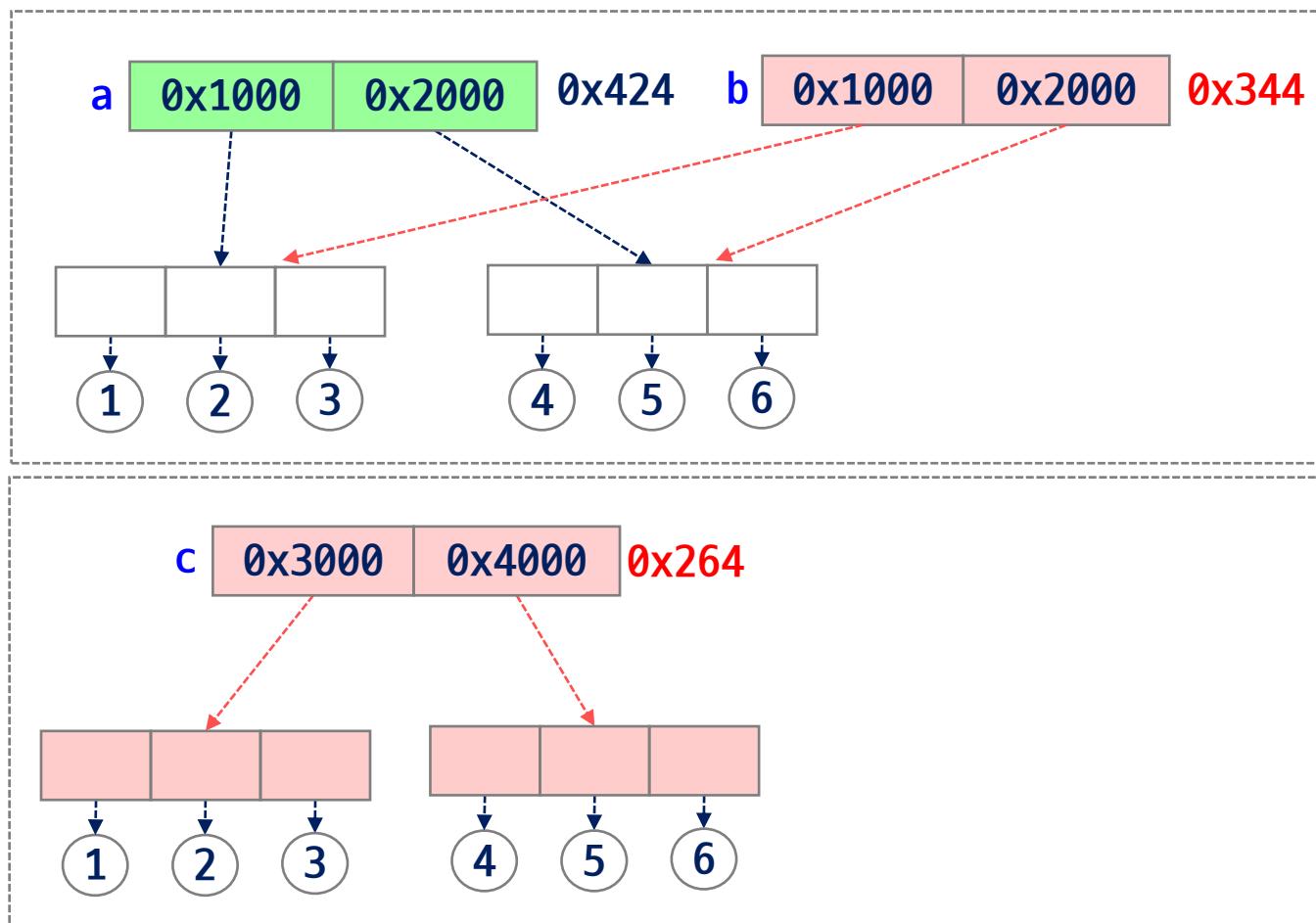
copy, deepcopy



a = [[1,2,3], [4,5,6]]

b = a.copy()

c = copy.deepcopy(a)



리스트(list) - item 정렬 메서드



정렬(sort)은 item을 특정 기준에 맞게 나열하는 작업이다

s.sort(*, key=None, reverse=False)

list 자체의 item 위치가 변경됨

s.sort()

오름차순 정렬 (숫자, 알파벳 대문자, 소문자, 한글)

s.sort(reverse=True)

내림차순 정렬, False : 오름차순

s.sort(key=len)

key : 비교 대상을 반환하는 함수/메서드를 지정함

S

“Are” “you” “happy?”

s.sort()

“Are” “happy?” “you”

s.sort(reverse=True)

“you” “happy?” “Are”

s.sort(key=len)

“you” “Are” “happy?”

“Are” “you” “happy?”

“숫자와 문자열이 섞여 있는 리스트는 정렬할 수 없다”

built-in 함수 - item 정렬



R = sorted(**iterable**, key=None, reverse=False)

VS

list의 sort 메서드

sorted 함수

대상

list 객체

iterable

반환

없음 (None)

새로운 list

예제

s.sort()
s.sort(reverse=True)
s.sort(key=len)

R=sorted(s)
R=sorted(s, reverse=True)
R=sorted(s, key=len)

문자열(str)의 구조 및 특징



☞ 문자열은 Text Sequence로 분류되며 **Immutable** 이다

특징

Sequence Type의 공통 연산 사용([not] in, +, *, subscription, slicing)

hashable 이므로 set의 item, dict의 key로 사용 가능

동등비교(==)연산은 모든 아이템이 순서까지 같을 때 참(True)

문자열의 아이템 변경(assignment) 및 삭제(deletion) 불가능

escape sequence

문자에 부여된 기능을 무시하고 그대로 출력 또는 다른 의미로 사용됨

escape sequence	설명
\'	'
\\"	"
\\\	\

escape sequence	설명
\t	[Tab]
\n	줄 바꿈
\newline	줄 바꿈 무시

문자열(str) - 대/소문자 변경, 확인 메서드



☞ s : str container

대/소문자 변경

R=s.upper()

대/소문자로 변경된 새로운 문자열 반환

R=s.lower()

문자 이외의 숫자나 공백문자는 그대로 사용됨

대/소문자 확인

R=s.isupper()

모든 글자가 대/소문자면 True, 아니면 False 반환

R=s.islower()

숫자, 공백 문자 등은 결과와 상관 없음(무시함)

s

apple135_A+

s.upper()

APPLE135_A+

s.lower()

apple135_a+

t

i am a girl!

t.isupper()

False

t.islower()

True

문자열(str) - 시작과 끝 확인 메서드



▶ `startswith`, `endswith` 는 특정 문자열로 시작 또는 끝 나는지 확인에 사용한다

`s.startswith(t [, start[, stop]])`

`s.endswith(t [, start[, stop]])`

`s` 문자열이 `t` 문자열로 시작되면/끝나면 `True` 반환

비교 범위를 `start`, `stop` 으로 지정할 수 있음

`s`

“Kim Bob”

`R=s.startswith('Kim')`

`R=True`

`R=s.endswith('Bob')`

`R=True`

문자열(str) - 분리 메서드



👉 split 은 하나의 문자열을 여러 개의 문자열로 나누기 위해 사용한다

R=s.split(sep=None, maxsplit=-1)

R=s.rsplit(sep=None, maxsplit=-1)

R=s.split()

연속된 whitespace를 1개 구분자로 취급

R=s.split(' ')

한 개의 space를 1개 구분자로 취급 : 권장 되지 않음!

R=s.split(maxsplit=1)

분리 동작 최대 횟수 : 최대 maxsplit+1 개 item

s

“how are you?”

s.split()

[“how”, “are”, “you?”]

s.split(' ')

[“how”, “are”, “”, “”, “you?”]

t

“2020-12-25”

s.split(maxsplit=1)

[“how”, “are you?”]

t.split('-')

s.rsplit(maxsplit=1)

[“how are”, “you?”]

문자열(str) - 결합 메서드



join은 여러 개의 문자열을 하나의 문자열로 만들기 위해 사용한다

R = s.join(iterable)

iterable의 item은 문자열이어야 함

R=‘-’ .join(iterable)

iterable을 하나의 문자열로 연결

item과 item 사이에 ‘-’를 넣어 하나의 문자열로 연결

s

[“2020”, “12”, “25”]

R=“ ”.join(s)

R=“2020 12 25”

R=“-”.join(s)

R=“2020-12-25”

t

[2020, 12, 25]

R=“-”.join(t)

TypeError: sequence item 0: expected str instance, int found

세트(set)의 구조 및 특징



- set 은 Set Types에 포함되는 **Mutable** 이다
- set은 집합을 의미하며, 중복되지 않는 **hashable** 객체를 모아 놓은 것이다

생성 방법

{a, b, c} set() set(iterable) 주의사항 { }는 set 아님!

아이템 타입

hashable 객체 (set 자체는 hashable 아님)

특징

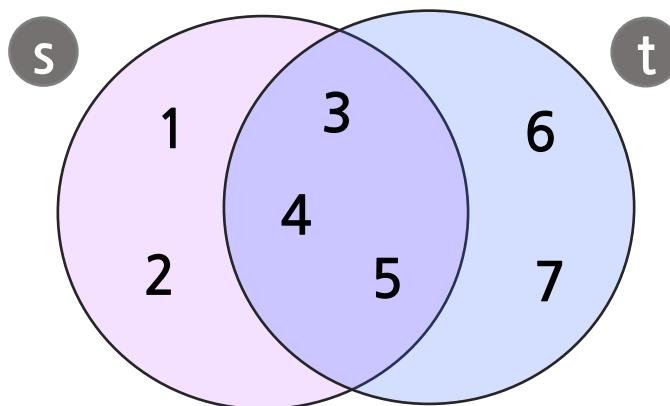
in, not in 연산 가능, subscription, slicing, +, * 연산 불가능

교집합, 합집합, 차집합, 대칭 차집합 연산 가능

동등비교(==)연산은 순서는 상관없이 아이템들이 같으면 참

set 은 아이템들의 중복 제거에 주로 사용됨

세트(set) - 집합 연산



s { 1, 2, 3, 4, 5 }
t { 3, 4, 5, 6, 7 }

합집합

여러 집합(s, t)의 원소를 합친 집합

{ 1, 2, 3, 4, 5, 6, 7 }

교집합

여러 집합(s, t)에 모두 포함 된 원소의 집합

{ 3, 4, 5 }

차집합

하나의 집합(s)에만 포함된 원소의 집합

{ 1, 2 }

대칭차집합

두 차집합(s-t, t-s)의 합집합

{ 1, 2, 6, 7 }

세트(set) - 집합 연산



☞ s, t : set container, it : iterable

합집합	s t ...	s.union(*it)	s { 1, 2, 3, 4, 5 }
교집합	s & t & ...	s.intersection(*it)	X (4, 5, 6)
차집합	s - t - ...	s.difference(*it)	s X : TypeError
대칭차집합	s ^ t	s.symmetric_difference(it)	s.union(X) : 가능!

s { 1, 2, 3, 4, 5 }	R = s t u	R = { 1, 2, 3, 4, 5, 6, 7, 9 }
t { 3, 4, 5, 6, 7 }	R = s & t & u	R = { 3, 5 }
u { 1, 3, 5, 7, 9 }	R = s - t - u	R = { 2 }
	R = s ^ t	R = { 1, 2, 6, 7 }

세트(set) - 집합 연산



set이 합집합, 교집합, 차집합, 대칭차집합 연산의 결과로 확장 되는 연산

$s |= t | \dots$

s를 s와 나열된 집합들의 합집합으로 확장

$s &= t \& \dots$

s를 s와 나열된 집합들의 교집합으로 확장

$s == t - \dots$

s를 s와 나열된 집합들의 차집합으로 확장

$s ^= t$

s를 s와 t 어느 한 곳에만 포함된 원소로 확장

s

{ 1, 2, 3, 4, 5 }

$s |= t$

$s = \{ 1, 2, 3, 4, 5, 6, 7 \}$

t

{ 3, 4, 5, 6, 7 }

$s &= t$

$s = \{ 3, 4, 5 \}$

세트(set) - item 추가 메서드



item 추가

s.add(x)

중복 값이면 무시됨, x는 hashable 이어야 함

x가 hashable이 아닌 경우 TypeError 발생

s

{ 1, 2, 3 }

s.add(5)

s = { 1, 2, 3, 5 }

t

(4, 5)

s.add(t)

s = { 1, 2, 3, (4, 5) }

u

[4, 5]

s.add(2)

s = { 1, 2, 3 }

s.add(u)

TypeError

세트(set) - item 삭제 메서드



특정 item 삭제

s.remove(x)

x가 s에 포함되어 있지 않으면 **KeyError** 발생

임의 item 삭제

s.pop()

임의의 아이템을 반환/제거

item이 없으면 **KeyError** 발생

모든 item 삭제

s.clear()

s

{ 1, 2, 3 }

s.remove(2)

s = { 1, 3 }

t

set()

s.remove(10)

KeyError

“set의 slot이 변경 됨”

s.discard(2)

s = { 1, 3 }

s.discard(10)

s = { 1, 2, 3 }

R = s.pop()

s = { 2, 3 }

R

1

R = t.pop()

KeyError

s.clear()

empty set

사전(dict)의 구조 및 특징



☞ 사전(dict)은 Mapping으로 분류되며 **Mutable** 이다

생성 방법

- {K:V} { K1:V1, K2:V2, ... } {} dict()
- dict(**kwarg) dict(mapping, **kwarg)
- dict(iterable, **kwarg)

아이템 타입

- **key** : 사전을 위한 index로 사용, **hashable** 객체만 사용 가능
- **value** : 제한 없음, 중복 사용 가능

특징

in, not in 연산 가능, subscription, slicing, +, * 연산 불가능

동등비교(==)연산은 순서 상관없이 item 들이 같으면 True

index를 사용하여 value를 참조/갱신하는 연산을 많이 함

사전(dict) - value 반환/갱신, item 추가/삭제 연산

☞ d : dict, k : key, v : value

value 반환

d[key]

key에 해당하는 value 반환, key 없는 경우 **KeyError**

value 갱신
item 추가

d[key]=value

존재하는 key : value 갱신
존재하지 않는 key : item 추가

item 삭제

del d[key]

key에 해당하는 item 삭제, key 없는 경우 **KeyError**

d

{ "name": "Tom", "age": 10 }

R = d['age']

R = 10

R = d['weight']

KeyError

d['age'] = 11

{ "name": "Tom", "age": 11 }

d['height'] = 140

{ "name": "Tom", "age": 10, "height": 140 }

del d['age']

{ "name": "Tom" }

del d['weight']

KeyError

사전(dict) - 비교 연산



☞ d : dict, k : key, v : value

포함 여부 확인

k in d

사전에 있는 key인지 확인, 있을 때 True

k not in d

사전에 없는 key인지 확인, 없을 때 True

동등 비교

==

순서에 관계 없이 모든 item 이 같을 때 True

in/ not in 연산에서 d.keys()와 d는 동일 의미를 갖음

d

{ "name" : "Tom", "age" : 10 }

t

{ "age" : 10, "name" : "Tom" }

'age' in d

True

'weight' not in d

True

d == t

True

사전(dict) - dict 객체 생성 메서드



key로 사용할 iterable 목록과 동일한 초기값을 사용한 객체 생성

`dict.fromkeys(iterable [,v=None])`
`d.fromkeys(iterable [, v=None])`

iterable의 item을 key로 하고, value를 v로 하여 “새 사전 반환”

v가 없다면 None을 사용함

`d = dict.fromkeys(['A', 'B', 'C'], 0)`

`d = { 'A' : 0, 'B' : 0, 'C' : 0 }`

`d = dict.fromkeys(['A', 'B', 'C'])`

`d = { 'A' : None, 'B' : None, 'C' : None }`

사전(dict) - key, value, item 목록 반환 메서드



d

```
{ "name": "Tom", "age": 10 }
```

key 목록 반환

d.keys()

dict_keys(['name', 'age'])

value 목록 반환

d.values()

dict_values(['Tom', 10])

item 목록 반환

d.items()

dict_items([('name', 'Tom'), ('age', 10)])

“tuple”

dict_keys, dict_values, dict_items 타입

- iterable 객체, 출력 시 타입이 함께 표시됨
- list, tuple 등의 type으로 수정하여 출력

사전(dict) - value 반환 메서드



- d : dict, key : hashable, default : key가 없을 때 반환 값

value 반환

d.get(key, default=None)

key가 없을 때의 값을 지정할 수 있음

default 지정없이, key 없을 때 None 반환

d { "name": "Tom", "age": 10 }

R = d.get("age", -1)

R = 10

R = d.get("weight", -1)

R = -1

R = d.get("weight")

R = None

사전(dict) - item 추가/갱신 메서드



☞ d : dict, key : hashable, defaultValue : 임의 객체

item 추가

d.setdefault(key, defaultValue)

key 값이 존재하지 않을 때 item 추가

item 갱신

d.update([t])

- t가 제공하는 key, value 쌍으로 사전 갱신
- t는 dict 객체이거나 iterable 사용
- iterable은 길이 2인 item들로 구성 되어야 함

d

{ "name": "Tom", "age": 10 }

t

{ "name": "Jerry", "height": 140 }

d.setdefault('name', 'Jerry')

추가 없음

d.setdefault('height', 140)

d = { 'name': 'Tom', 'age': 10, 'height': 140 }

d.update(t)

d = { 'name': 'Jerry', 'age': 10, 'height': 140 }

사전(dict) - item 제거 메서드



👉 d : dict, key : hashable, default : key가 없을 때 반환 값

value 반환/
item 제거

d.pop(key,
[, default])

key가 없을 때의 값을 지정할 수 있음

default 지정 없이, key 없으면 **KeyError** 발생

item 반환/
item 제거

d.popitem()

- 마지막에 추가된 item 반환 및 제거
- 반환된 item은 (key, value)의 tuple 객체

▪ d가 empty dict 객체인 경우 **KeyError** 발생

모든 item 제거

d.clear()

d

{ “name” : “Tom”, “age” : 10 }

R = d.pop(“name”, -1)

R = “Tom”

d = {‘age’ : 10}

R = d.pop(“weight”)

Key Error

R = d.popitem()

R = (“age”, 10)

d = {‘name’ : ‘Tom’}

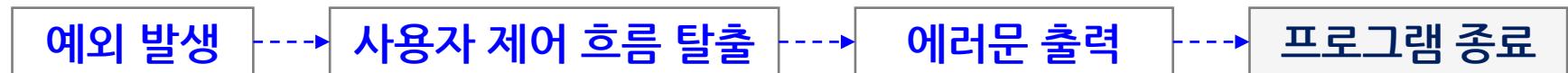
d.clear()

d = { }

Exception



- Exception은 예외상황(오류) 발생시 프로그램의 제어 흐름을 조정하기 위한 event이다



```
d = {'A': 1, 'B': 2, 'C': 3}  
a = d['D']
```

→ KeyError : 존재하지 않는 key를 사용

```
try:  
    a = d['D']
```

→ 예외 발생 가능한 문장을 작성한다

```
except:
```

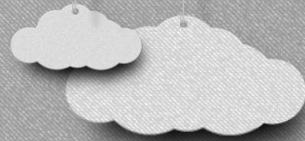
```
    import traceback  
    print(traceback.format_exc())  
    a = -1
```

→ 예외 처리를 위한 문장을 작성한다

```
    print(a)
```

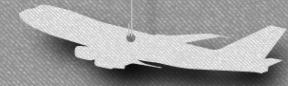
```
b = d.get('D', -1)  
print(b)
```

→ 예외 발생시 처리 방법을 제공하는 메서드를 사용



04

Compound statements - if, for, while



Line structure



python 프로그램은 여러 개의 논리적인 행들로 나뉜다

Logical Lines	<ul style="list-style-type: none">Logical Line의 끝은 NEWLINE 토큰으로 표현됨행 결합 규칙에 따라 하나 이상의 Physical Line으로 구성 가능
Physical Lines	<ul style="list-style-type: none">ENTER를 누르면 만들어지는 하나의 행
Comment	<ul style="list-style-type: none">#으로 시작하며 Logical Line을 종료 시킴
명시적 행 결합	<ul style="list-style-type: none">₩ 기호를 이용하여 하나의 Logical Line으로 결합₩ 기호는 주석을 결합하지 못하므로₩ 뒤에 주석 사용할 수 없음
묵시적 행 결합	<ul style="list-style-type: none">0, [], {} 가 사용 되는 표현은₩ 없이도 결합이어지는 줄의 들여쓰기는 중요하지 않음묵시적 행 결합하는 행 사이에는 NEWLINE 토큰이 생성되지 않음주석을 포함할 수 있음
Blank lines	<ul style="list-style-type: none">공백, 탭, 폼피드(₩f) 및 주석으로만 구성된 Logical Line은 NEWLINE 토큰이 만들어지지 않음

문(statement)



statement는 실행가능한(executable) 최소의 독립적인 코드이다

keyword 아님

3부에서 살펴볼 내용

simple statement

expression statements

assignment

del

global

nonlocal

assert

yield

raise

import

continue

break

return

pass

compound statement

if

조건에 따른 분기를 위한 구문

while

조건에 따른 반복을 위한 구문

for

지정된 횟수 만큼 반복을 위한 구문

try

예외 처리를 위한 구문

with

리소스 핸들을 자동 반환처리 되도록 하는 구문

def

함수 정의

class

클래스 정의

simple statement

한 개의 Logical Line으로 구성, 세미콜론으로 여러 개 나열 가능

compound statement

다른 statements 를 포함

다른 statements 의 실행을 제어하거나 영향을 줌

https://docs.python.org/3/reference/simple_stmts.html

https://docs.python.org/3/reference/compound_stmts.html

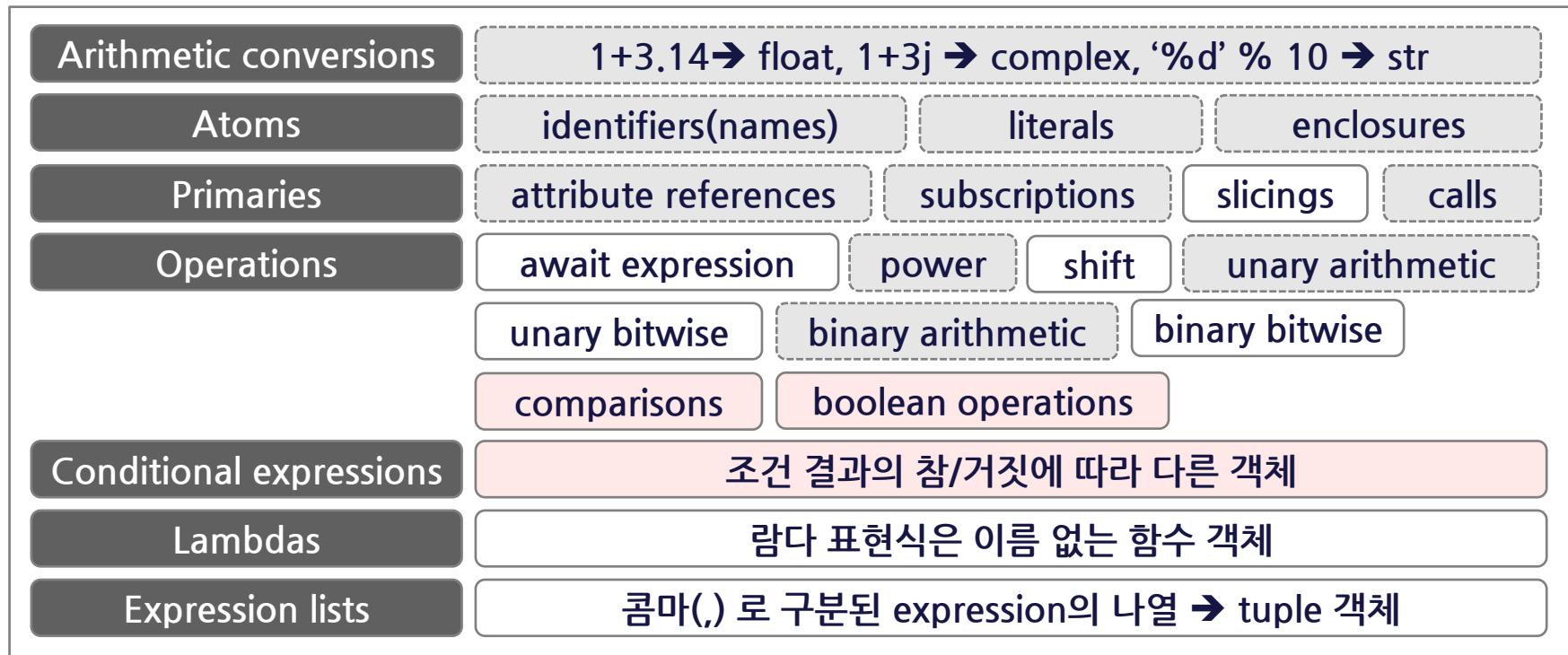
식(expression)



expression은 한 개의 객체로 evaluate 될 수 있는 것을 의미한다

→ 이름에 바인딩 가능

3부에서 살펴볼 내용



- comparisons: value, membership test(in, not in), identity comparisons(is, is not)
- enclosure: parenthesized forms, {list, set, dict} displays, {generator, yield} expressions

<https://docs.python.org/3/reference/expressions.html>

BNF(Backus-Naur Form)



- BNF는 문맥 자유 문법을 나타내기 위해 만들어진 표기법이다

메타 문자	의미
::=	대입 또는 대치
"상수"	상수는 쌍 따옴표로 묶어 표현
"A"..."B"	A~B까지 상수의 연속된 나열
()	그룹
	나열된 요소 중 하나 (or)

메타 문자	의미
[]	선택적으로 사용 가능한 요소
*	0 개 이상의 요소 (최소 0개)
+	1개 이상의 요소 (최소 1개)
+n	n개의 연속된 동일 요소

- 예) 16진수(hexinteger)에 대한 BNF 표기법

```
hexinteger ::= "0" ("x" | "X") (["_"] hexdigit)+  
hexdigit   ::= digit | "a"..."f" | "A"..."F"  
digit      ::= "0"..."9"
```

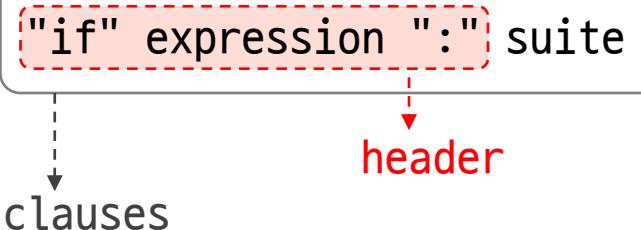
0x13AB78EF
0x13_AB_78_EF
0x13AB_78EF
0xA__B__C
0xAB_

BNF - if statement



- if 문은 조건부 실행에 사용된다

```
if_stmt ::= "if" expression ":" suite  
          ("elif" expression ":" suite)*  
          ["else" ":" suite]
```



- suite는 절에 의해 제어되는 statement 그룹이다

```
suite      ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEENT  
statement  ::=  stmt_list NEWLINE | compound_stmt  
stmt_list  ::=  simple_stmt (";" simple_stmt)* [";"] 동일 level의  
                           indentation
```

Example - if statement



if expression :

expression이 참인 경우 실행 문장

if a > b:

msg = "a > b"

if expression :

expression이 참인 경우 실행 문장

else :

expression이 거짓인 경우 실행 문장

if a > b:

msg = "a > b"

else:

msg = "a <= b"

if expression1 :

expression1이 참인 경우 실행 문장

elif expression2 :

expression1은 거짓이고,

expression2가 참인 경우 실행 문장

elif expression3 :

...

else :

모든 expression이 거짓인 경우 실행 문장

if a > b:

msg = "a > b"

elif a < b:

msg = "a < b"

else:

msg = "a == b"

Example - if statement



if expression : simple statement1; simple statement2; ...

```
if a > b: msg = "a > b"; msg2 = "b < a"
```

if expression :
 simple statement
 simple statement1; simple statement2; ...

```
if a > b:  
    msg = "a > b"; msg2 = "b < a"
```

if expression :
 compound statement

```
if a > b:  
    if b > c : msg = "a > b > c"
```

pass 문



- ▶ pass 문은 실수(error)로 비워져 있지 않다는 것을 명시할 때 사용한다
- ▶ compound statement (if, for, while, try, with, def, class)의 suite를 empty로 처리 할 때 사용한다
 - compound statement는 1개 이상의 statement가 필요하다
 - 의도적 empty로 둔 것을 명시하여야 SyntaxError가 발생하지 않는다

```
[class A:  
    pass]
```

```
a = A()
```

```
class B:  
    [ def func(self):  
        pass]
```

```
b = B()  
b.func()
```

```
[def func():  
    pass]
```

```
func()
```

```
a = 500  
if a > 1000:  
    print('large')  
elif a < 100:  
    print('small')  
else:  
    pass
```

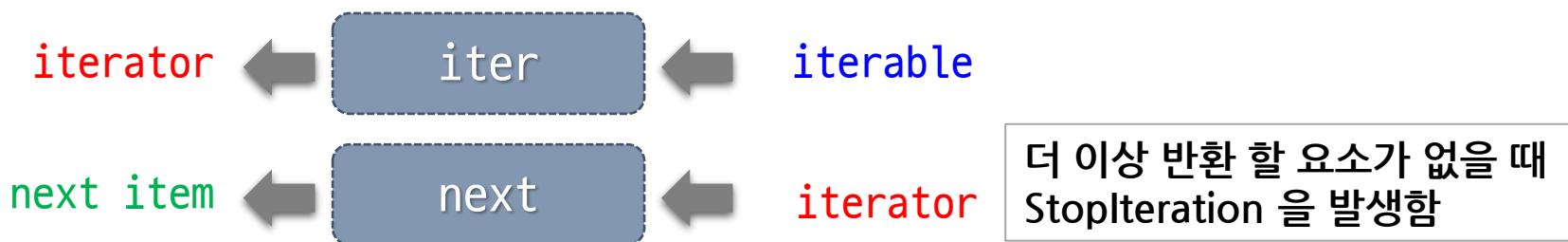
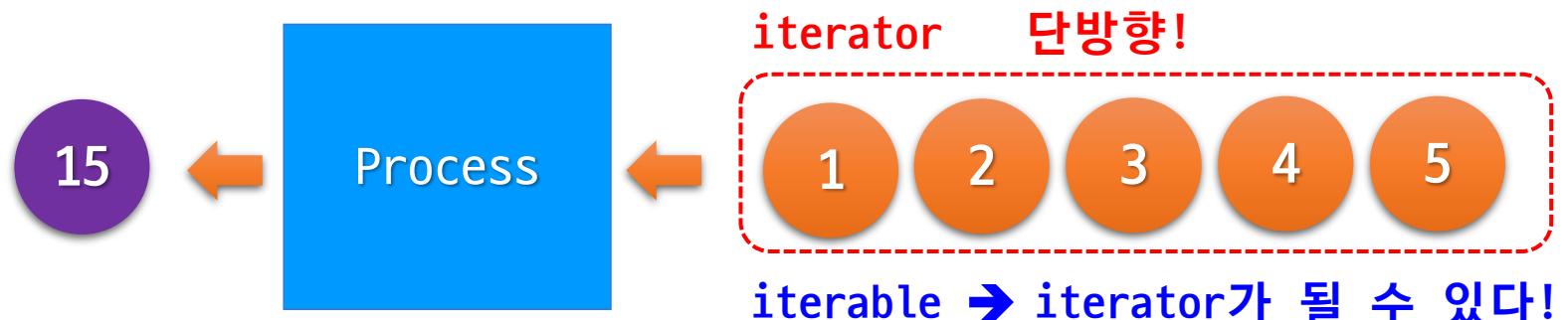
iteration(반복)



왜 iteration이 필요할까?



어떻게 iteration을 수행할까?



iterable, iterator



☞ iterable, iterator 개념을 정리해 보자

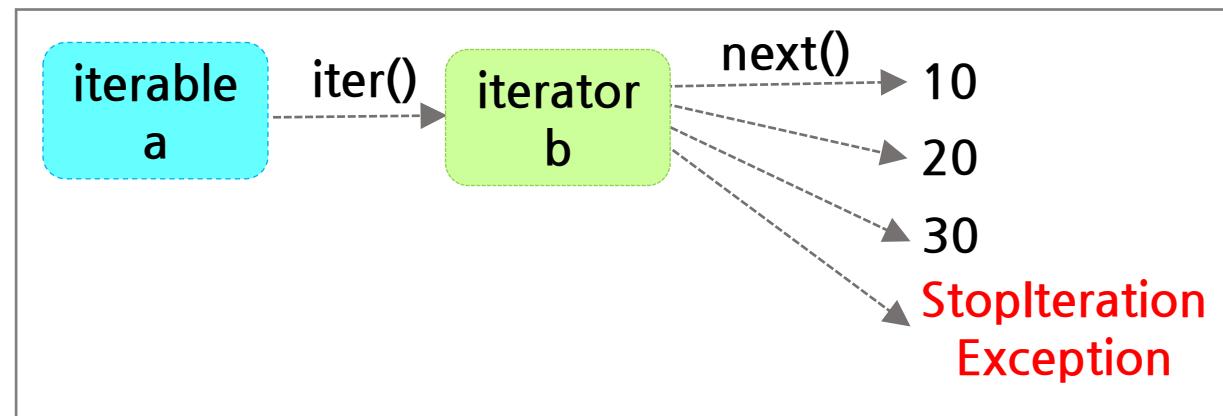
iterable

iter()의 argument로 사용 가능 객체, list, tuple, str, set, dict, range 등

iterator

next() 및 iter() 의 argument로 사용 가능 객체 “Iterator is Iterable”

```
a = (10, 20, 30)
b = iter(a)
print( next(b) )
print( next(b) )
print( next(b) )
print( next(b) )
```

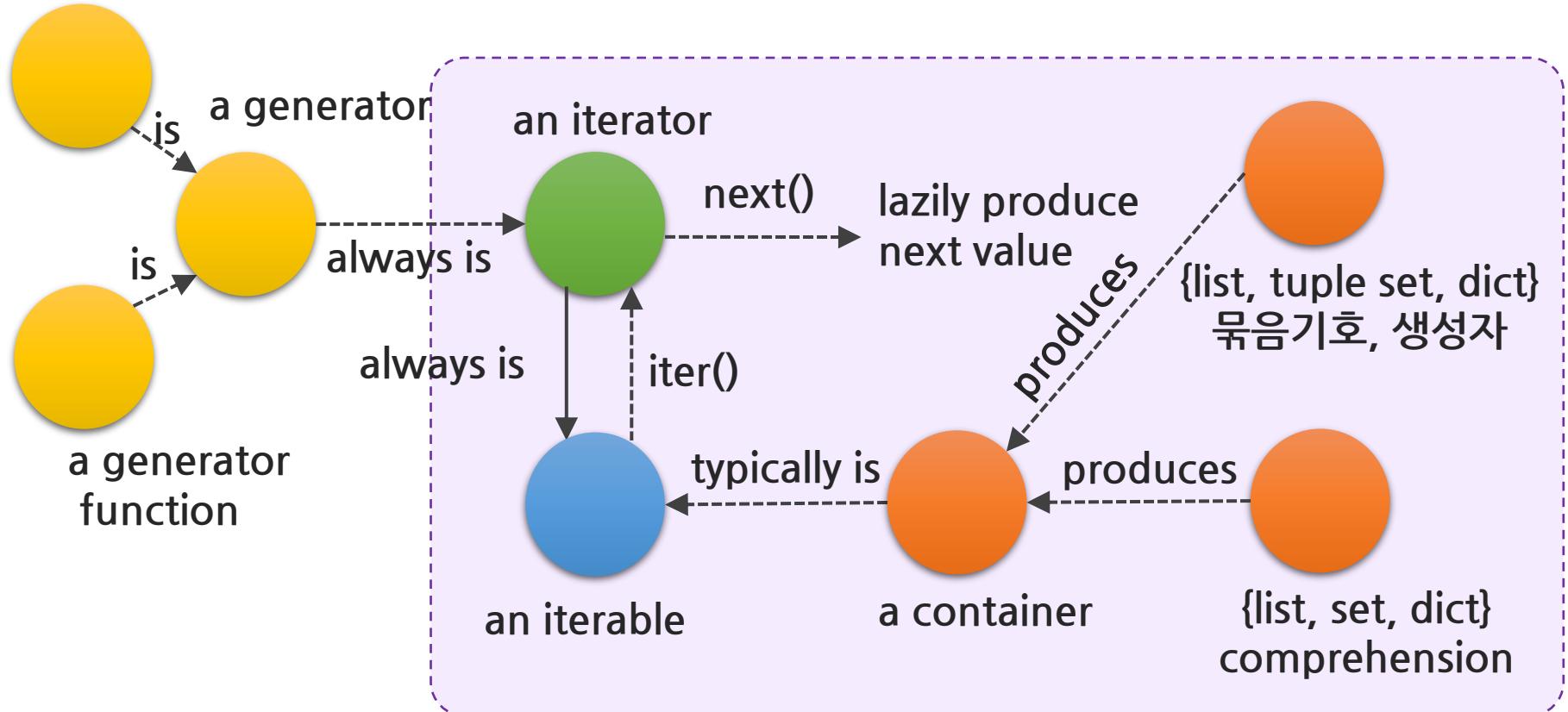


```
for x in (10, 20, 30):
    print(x)
```

iterator's family!



a generator expression



BNF - for statement



for 문은 iterable 객체의 아이템을 순회할 때 사용한다

```
for_stmt      ::=  "for" target_list "in" expression_list ":" suite  
                      ["else" ":" suite]  
expression_list ::=  expression ("," expression)* [","]
```

```
for x in [1,2,3,4]:  
    print(x)  
        ↓  
    tuple
```

```
for x, y in [(1,2), [3, 4]]:  
    print(x, y)  
        ↓  
    tuple
```

- expression_list는 한 번만 값이 구해지며 결과는 **iterable 객체**이어야 한다
- expression_list 결과로 iterator가 만들어진다
- iterator의 순환마다 각 항목이 target_list에 대입되고 suite가 실행된다
- iterator의 다음 항목이 없으면 else절의 suite를 실행하고 반복을 종료한다
- 첫 번째 suite에서 실행되는 break문은 else절을 실행하지 않고 반복을 종료한다

break, continue 문



▶ break, continue는 반복 동작의 제어를 위해 사용한다

break

else절을 실행하지 않고 반복을 종료

continue

나머지 부분을 건너뛰고 다음 항목을 변수에 넣는 작업으로 넘어감

```
for x in (1, 2, 3):
    if not x % 2 : break
    print(x)
else:
    print("done1")
```

```
for x in (1, 2, 3):
    if not x % 2 : continue
    print(x)
else:
    print("done2")
```

function : range



- range는 immutable sequence of numbers type 이다

range(stop)

0부터 1씩 증가하는 stop 보다 작은 정수로 구성

range(start, stop)

start부터 1씩 증가하는 stop 보다 작은 정수로 구성

range(start, stop, step)

start부터 step씩 증가/감소하는 stop 보다
작은/큰 정수로 구성 (step은 0일 수 없음)

- range 함수는 iterable 객체인 range 객체를 반환한다

list, tuple 보다 작은 메모리 사용

Sequence 이므로 subscription 할 수 있음

BNF - while statement



☞ while 문은 식이 참인 동안 반복 수행하기 위해 사용한다

```
while_stmt ::= "while" expression ":" suite  
           ["else" ":" suite]
```

```
a = 0  
while a<3 :  
    print(a)  
    a+=1
```

```
while True :  
    a = int(input())  
    if not a : break  
    print(a)
```

- expression이 참이면 첫 번째 suite를 실행한다
- expression이 거짓이면 else 절의 suite가 있을 경우 실행하고 반복을 종료한다
- 첫 번째 suite에서 실행되는 break문은 else절을 실행하지 않고 반복을 종료한다
- expression이 처음부터 거짓일 수 있으며, 이때는 else가 있는 경우 else만 실행된다
- **while True :**를 사용하면 무한반복문이 된다

Example - while statement



```
while expression : simple statement1, simple statement2 ...
    simple statement1; simple statement2 ...
```

```
a = 0
while a<3 : print(a); a+=1
```

```
while expression :
    simple statement1; simple statement2 ...
```

```
a = 0
while a<3 :
    print(a)
    a+=1
```

```
while expression :
    compound statement
```

```
while True :
    a = int(input())
    if not a : break
    print(a)
```

Example - while statement



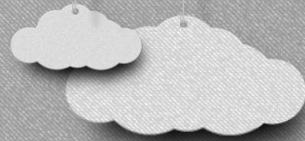
```
while expression :  
    statement1  
    if expression : break  
    if expression : continue  
    statement2  
else :  
    statement3
```

```
tot = 0  
while True :  
    a = int(input())  
    if a <= 0 : break  
    tot += a  
else: print("done")  
print(tot)
```

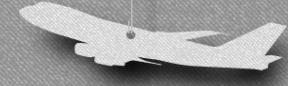
1
5
10
0
16

```
tot = 0  
while tot<100 :  
    a = int(input())  
    if a <= 0 : continue  
    tot += a  
else: print("done")  
print(tot)
```

25
-10
50
0
30
done
105



05



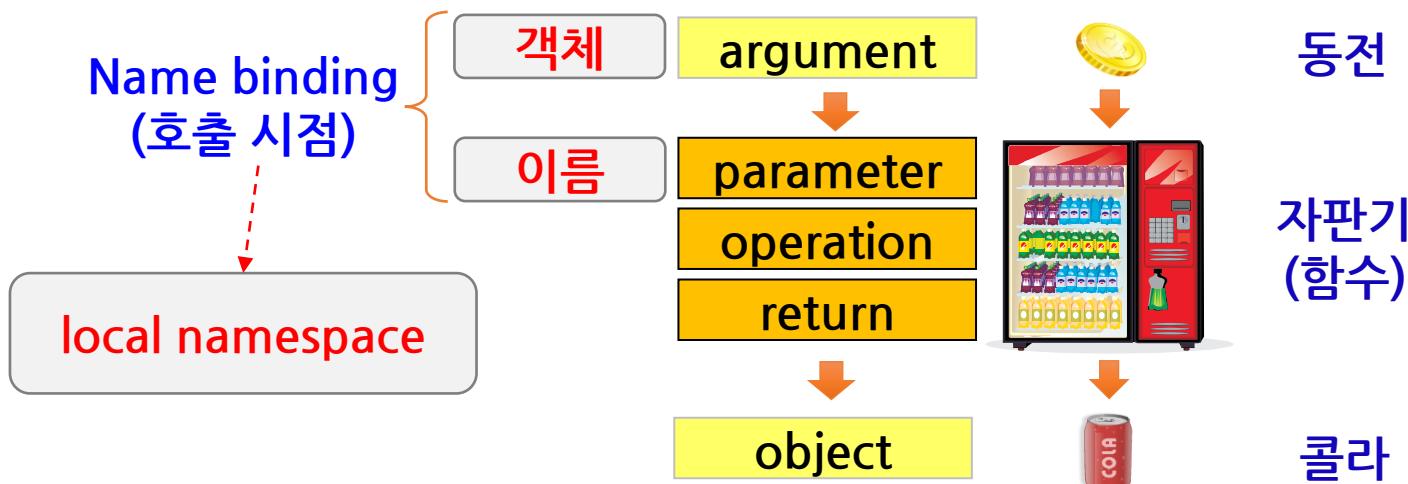
Function의 이해 및 활용



함수(function)



- 함수란 특정 기능 수행을 위한 명령의 집합이다
- 입력, 처리, 출력으로 구성된다.



함수 생성 및 호출 과정 이해



▣ 함수 생성 및 호출 과정은 다음과 같은 순서로 발생합니다

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

```
r = add(10, 20)
```

```
def function_name ( parameter_list ) : suite
```

```
function_name ( argument_list )
```

global Namespace

add : 0x100

r : 0x400

local Namespace

a : 0x200

b : 0x300

c : 0x400

Memory

code : function 0x100

10 : int 0x200

20 : int 0x300

30 : int 0x400

co_varnames, bytecode ...

argument의 종류와 사용법



- argument는 함수를 호출 시 함수에 전달하는 객체이다

positional

위치에 따른 객체 전달을 하는 argument

keyword

이름에 따른 객체 전달을 하는 argument

- argument rule

Positional과 Keyword를 섞어 사용할 수 있다

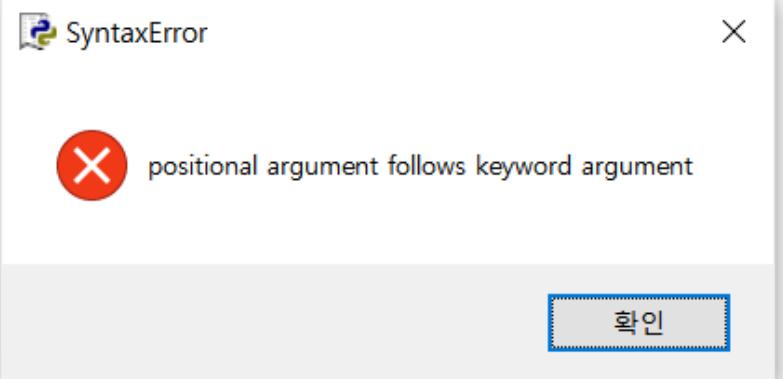
다만, Keyword를 Positional의 앞에 사용할 수 없다

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

x = add(10, 20)           ➔ Positional
y = add(a=10, b=20)       ➔ Keyword

print(x, y)
```

add(a=10, 20) ➔ SyntaxError



argument 전달 방법



(argument는 함수를 호출(call)하면서 넘겨주는 객체이다)

test [comp_for]	func(x)	객체(object) 전달
	func(x for x in range(5))	generator 객체 전달
test := test	func(x := 10)	x를 namespace에 저장 및 객체 전달
test = test	func(x = 10)	x라는 parameter에 객체 전달
** test	func(**x)	x가 dict 객체일때 unpack해서 전달
* test	func(* x)	x를 unpack해서 전달

parameter의 종류



parameter는 argument 전달 방식에 따라 다섯 가지로 종류로 구분된다

positional or keyword

positional 또는 keyword argument로 객체를 전달 받음

positional only

positional argument로만 객체를 전달 받음

keyword only

keyword argument로만 객체를 전달 받음

var-positional

0개 이상, 임의의 개수 positional argument로 객체를 전달 받음

var-keyword

0개 이상, 임의의 개수 keyword argument로 객체를 전달 받음

```
def func1(a, b): pass  
def func2(a, /): pass # v3.8  
def func3(*, a): pass  
def func4(*a, b): pass  
def func5(**a): pass
```

```
func1(10, b=10)  
func2(10)  
func3(a=20)  
func4(10, 20, 30, 40, b=20)  
func5(x=10, y=20, z=30)
```

parameter의 default argument 지정



- 함수 정의 시 parameter에 “기본 객체(default argument)”를 지정할 수 있다
 - parameter = expression의 형태로 지정한다

가변 parameter들은 기본 객체를 지정할 수 없다

‘기본 객체’가 있는 경우는 argument 전달을 하지 않을 수 있다

함수 호출 시 argument 전달이 없을 때 ‘기본 객체’가 사용된다

‘기본 객체’가 지정되지 않은 경우 반드시 argument 전달을 해야 한다

```
def func1(a, b=20): pass  
def func2(*, a=10, b): pass  
def func3(*a, b, c=30): pass
```

```
func1(10)  
func2(b=20)  
func2(a=30, b=20)  
func3(b=20)  
func3(b=20, c=40)
```

parameter 연습



- ▣ 다음 함수의 각 parameter 의 종류를 표기하여 보도록 한다
- ▣ pPK, pPKd, pK, pKd, pVP, pVK 중 하나를 선택한다

함수 prototype	parameter 종류, 최소 argument 개수
func1(a, b, c)	▪ a : pPK, b : pPK, c : pPK, 3개
func2(a, b, c=0)	
func3(a, *, b=0, c=None)	
func4(*, a=0, b=0)	
func5(a, b, *c)	
func6(*a, b, c=0)	
func7(a, b=2, **c)	
func8(*a, **b)	

컨테이너의 unpack 1/3



☞ 자료를 나누어 사용하는 것을 unpack 이라고 한다

☞ iterable, iterator, generator, file 등은 unpack 처리가 가능하다

- list, tuple 등은 index로 데이터에 접근하기 때문에 의미를 부여하기 힘들다
- 의미 부여를 위해 unpack을 사용하면 유용하다

```
pf = ['Julie', 42, 160, 'A']
print(pf[0], pf[1], pf[2], pf[3])
```

```
name, age, height, bType = pf      } unpack
print (f'My name is {name}, I am {age} years old.')
print (f'My height is {height} and BloodType is {bType}')
```

```
Julie 42 160 A
My name is Julie, I am 42 years old.
My height is 160 and BloodType is A
```

컨테이너의 unpack 2/3



☞ *, ** 을 사용한 argument unpack을 살펴보자

☞ *, ** 는 argument 위치에서 unpack을 수행한다

- *는 tuple, list, set 등을 ‘item1, item2, ...’의 형태로 unpack 한다
- **는 dict 를 ‘key1=value1, key2=value2, ...’ 형태로 unpack 한다

```
mytuple = ('Apple', 'Red', 600)
mydict  = {'name':'Apple', 'color':'Red', 'price':600}

print('{0} : price is {2}, color is {1}'.format(*mytuple))
                           'Apple', 'Red', 600
print('{name} : price is {price}, color is {color}'.format(**mydict))
                           name='Apple', color='Red', price=600
```

dict 객체에 '*'를 한 개만 붙이면 콤마로 구분된 key의 목록이 된다

컨테이너의 unpack 3/3



▣ 다음 예제를 통해 * 의 사용법을 알아보자

▣ 컨테이너 자료 중 일부만 unpack이 필요할 때는 어떻게 할까?

- 모든 자료를 unpack하여 저장하면 불편할 때가 있다
- 이때, * 를 사용하여 원하는 자료만 unpack 하여 사용한다

```
scores = [ 80, 80, 30, 80, 70, 70, 70, 90 ]
scores.sort()
first, *middle, last = scores
print(type(middle), middle)
avg = sum(middle)/len(middle)
print(avg)
```

첫 번째와 마지막 아이템을 제외한
나머지 아이템들을 middle에 저장한다

```
<class 'list'> [70, 70, 70, 80, 80, 80]
75.0
```

Namespace의 종류 및 특징



크게 두 가지로 분류되며 접근 체계(Name Resolution)가 다르다



class namespace

class의 method, attribute 이름 관리

instance namespace

instance의 method, attribute 이름 관리



built-in namespace

기본 built-in 함수, 타입 및 예외 이름 관리

global namespace

global variable 관리, module별 존재

local namespace

함수/메서드 별로 존재, local variable 관리

Namespace 찾기!



▣ 다음 코드 실행에 몇 개의 namespace가 사용되었을까요?

```
a = 100
b = 200

class A:
    def __init__(self, value):
        self.value = 10

    def add(self, other):
        return self.value + other

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

x = A(10)
print(x.add(20))
print(add(a, b))
```

우선 ‘이름’을 모두
찾아 보세요!

‘이름’이 포함되는
namespace를 그려보세요!

전역변수, 지역변수



- ▶ 전역변수는 global namespace, 지역변수는 local namespace에 등록된 이름이다

global variable
(전역변수(이름))

- 하나의 모듈 전체에 영향을 주는 이름
- 모듈 level에서 생성 되며, global namespace 에 등록

local variable
(지역변수(이름))

- 하나의 함수/메서드(블록)에만 영향을 주는 이름
- 블록 내에서 생성 되며, local namespace 에 등록

- ▶ Name Resolution은 이름 검색 규정이다

읽기

현 Scope부터 상위로
이동하며 이름 검색

이름이 없으면 **NameError**
객체에 바인딩 되어 있지 않으면 **UnboundLocalError**

하위 local
namespace

상위 local
namespace

global
namespace

built-in
namespace

NameError

쓰기

현 Scope의 namespace에서 검색

이름이 없으면 **생성**, 있으면 **갱신**

keyword : global, nonlocal



global, nonlocal은 변수 앞에 붙이는 키워드로 사용 namespace를 제한한다

global

global namespace 사용

nonlocal

자기 상단 local namespace 사용

언제 global, nonlocal 키워드를 사용할까?

다른 영역의 변수를

Read/Write 용도로 사용하기 위해

global

지역 영역에서 전역 변수

nonlocal

지역 영역에서 바깥 지역 변수

```
a = 100 -----> global
def outerFunction():
    a = 200 -----> nonlocal
    def innerFunction():
        a = a + 1
        local
        innerFunction()
        print(a)

outerFunction()
```

global



```
a = 100
def myFunction():
    a = a + 1

myFunction()
print(a)
```

```
a = 100
def myFunction():
    global a
    a = a + 1

myFunction()
print(a)
```

‘a’ : local namespace에 등록

UnboundLocalError 발생

‘a’: global namespace에 있는 것 사용(Read/Write)

‘a’가 101 값을 갖는 정수를 binding 함

“권장하지 않습니다”

Container 사용 권장!

```
a = [100]
def myFunction():
    a[0] += 1
```

```
myFunction();
print(a[0])
```

```
d = {'a' : 100}
def myFunction():
    d['a'] += 1
```

```
myFunction();
print(d['a'])
```

lambda 함수는 언제 사용할까?



- 이름 없는 function을 ‘한 줄’로 작성할 수 있는 ‘expression’ 작성 방법이다

```
def add1(a, b):  
    return a + b  
x = add1(10, 20)
```

```
add2 = lambda a, b : a + b  
y = add2(10, 20)  
  
y = (lambda a, b : a + b)(10, 20)
```

한 줄의 간단한 함수가 필요한 경우

프로그램의 가독성을 높이기 위해

주로 다른 함수의 argument로
간단한 동작을 하는 함수 전달 시 사용

lambda 함수의 특징



```
def function_name ( parameter_list ) : suite
```

```
lambda parameter_list : expression
```

def 함수	lambda 함수
<pre>def add(a, b): return a + b x = add(10, 20)</pre>	<pre>y = (lambda a, b : a + b)(10, 20)</pre>
statement	expression
statement, expression을 모두 사용	expression만 사용
return문을 사용하여 결과 반환	expression의 실행 결과가 반환됨 여러 개 반환 시 Container로 반환
namespace에 name이 등록됨	namespace에 name이 등록되지 않음

람다의 작성 예

▣ 다음 람다의 예를 실행하여 작성법을 알아본다

```
print( (lambda x : x**2)(10)) -----> 바로 사용  
power = lambda x: x**2  
print (power(10)) -----> 이름에 저장하여 사용
```

```
print( (lambda x, y : x+y)(10, 20)) -----> parameter가 두 개인 경우  
add = lambda x, y: x+y  
print(add(10,20))
```

```
ex = lambda : 10 -----> parameter를 사용하지 않는 경우  
print(ex())
```

lambda w : 100 처럼 parameter를 지정하고도
사용하지 않을 수 있음 (항상 100 반환)

```
y = 20  
ex = lambda x : x+y -----> 외부 이름(y)을 사용한 경우  
print(ex(10))
```

Python built-in 함수 목록



파이썬의 built-in 함수 목록은 다음과 같다

Built-in Functions					
abs()	all()	any()	ascii()	bin()	bool()
breakpoint()	bytearray()	bytes()	callable()	chr()	classmethod()
compile()	complex()	delattr()	dict()	dir()	divmod()
enumerate()	eval()	exec()	filter()	float()	format()
frozenset()	getattr()	globals()	hasattr()	hash()	help()
hex()	id()	input()	int()	isinstance()	issubclass()
iter()	len()	list()	locals()	map()	max()
memoryview()	min()	next()	object()	oct()	open()
ord()	pow()	print()	property()	range()	repr()
reversed()	round()	set()	setattr()	slice()	sorted()
staticmethod()	str()	sum()	super()	tuple()	type()
vars()	__import__()				

<https://docs.python.org/3/library/functions.html?#built-in-functions> 참조

모듈(module)



모듈(=library) 사용으로 빠르고 편리하게 프로그램을 작성할 수 있게 된다

built-in 함수, 모듈
(설치 불필요)



외부 모듈 (설치 필요)



빠르고 편한 프로그래밍

sys
pprint
os
pickle
shelve
glob
itertools
copy
...

데이터 수집	BeautifulSoup, selenium
데이터 처리	numpy, pandas
ML, DL	Scikit learn, tensorflow, keras
이미지 처리	pillow, OpenCV
시각화, 그래프	matplotlib, seaborn, bokeh
HTTP, 웹페이지	requests, django
자연어 처리	NLTK, TextBlob
...	...

<https://www.ubuntupit.com/best-python-libraries-and-packages-for-beginners/>

모듈과 패키지



모듈 사용을 위해서는 이름을 namespace에 등록하는 import 가 필요하다

모듈(module)은 하나의 파일 형태로 작성되며 XX.py로 저장된다

- 특정 모듈 또는 모듈내 특정 정보 import 방법

import 모듈이름1 [as 모듈닉네임1] [, 모듈이름2 [as 모듈닉네임2]] ...

from 모듈이름 import 사용할 모듈 내의 특정정보 or *

- 모듈 내부의 함수는 ‘모듈이름.함수명’ 또는 ‘모듈닉네임.함수명’ 형식으로 사용한다
- from을 사용하면 모듈이름을 생략하고 함수이름으로 바로 호출 할 수 있다
 - from ... import * 형식의 import는 '_'로 시작하는 이름을 제외한 모든 이름을 연결한다

패키지(package)는 모듈을 모아 놓은 디렉터리이다

- 패키지내의 특정 모듈 import 방법

import 패키지이름.모듈이름 [as 모듈닉네임]

from 패키지이름 import [모듈이름1 [as 모듈닉네임1]] ...

from 패키지이름.모듈이름 import 사용할 모듈 내의 특정정보 or *

import 예시



- ▣ 다음 import 방법을 살펴 보자
- ▣ 패키지, 모듈, 함수를 구분하여 보도록 한다

```
import pprint as pp
from sys import getsizeof, getrefcount
import matplotlib.pyplot as plt1
from matplotlib import pyplot as plt2
from matplotlib.pyplot import plot, show

pp.pprint(globals())
print(getsizeof(1234567890))
print(getrefcount(1))
```

```
import pandas as pd
from [ ] import DataFrame
a = pd.DataFrame()
b = DataFrame()
print(a, b, sep='\n')
```

정상 동작을 위해 네모 안에
넣어야 하는 단어는 무엇인가?

```
'getrefcount': <built-in function getrefcount>,
'getsizeof': <built-in function getsizeof>,
'plot': <function plot at 0x0D308BB8>,
'plt1': <module 'matplotlib.pyplot' from [ ]>,
'plt2': <module 'matplotlib.pyplot' from [ ]>,
'pp': <module 'pprint' from 'C:\\\\Python37-32\\\\lib\\\\pprint.py'>,
'show': <function show at 0x0D2C73D8>}
```

plt1, plt2 의 경로는 삭제 했음

메인 모듈, 하위 모듈



▶ python은 main 함수가 없다

▶ 메인 모듈과 import 모듈의 모든 module-level 코드(들여쓰기 없는)를 실행한다

- 함수나 클래스는 객체가 생성되지만, 실행되지 않는다
- import 모듈의 실행되는 코드를 실행하고 싶지 않을 경우 __name__ 속성을 활용한다
- __name__에 저장된 값이 '__main__'인 경우에만 실행하도록 한다

```
if __name__ == '__main__': # if 문 활용  
    실행 코드
```

▶ __name__은 '현재 모듈'의 이름을 저장하는 속성이다

- python은 실행 방법에 따라 메인 모듈을 결정한다

직접 실행된 파이썬 모듈

메인 모듈

__name__에 '__main__'이 저장됨

import 된 파이썬 모듈

하위 모듈

__name__에 모듈이름이 저장됨

메인 모듈, 하위 모듈 확인



▣ 다음의 코드를 실행하여 결과를 분석하라

```
#mymodule1.py  
import mymodule2
```

mymodule1.py 를 실행 시

→ import 문에 의해 mymodule2.py 도 실행 됨

```
print('mymodule1 - name : ' + __name__)
if __name__ == '__main__':
    print ('main module!')
else:
    print (__name__ + ' is a sub module!')
```

```
mymodule2 - name : mymodule2
mymodule2 is a sub module!
mymodule1 - name : __main__
main module!
```

```
#mymodule2.py
```

mymodule2.py 를 실행 시 mymodule1.py은 실행 안됨

```
print('mymodule2 - name : ' + __name__)
if __name__ == '__main__':
    print ('main module!')
else:
    print (__name__ + ' is a sub module!')
```

```
mymodule2 - name : __main__
main module!
```

모듈(module) 만들어 사용하기



- 모듈을 import 하는 방법을 익혀보도록 한다
- 두 개의 서로 다른 파일로 작성하되, 동일 디렉터리에 위치하도록 한다
 - 파일을 만들면 그 파일의 이름이 모듈이름이 된다

```
import module1
```

```
a = 40
```

```
b = 10
```

```
print('%d + %d = %d' %(a, b, module1.add(a, b)))
```

```
from module1 import add
```

```
a = 40
```

```
b = 10
```

```
print('%d + %d = %d' %(a, b, add(a, b)))
```

```
#module1.py
```

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

```
def sub(a, b):  
    ret = a - b  
    return ret
```

- from module1 import * 을 사용하면 add, sub 둘 다 사용 가능하다
단, from ... import * 형식은 이름이 '_'로 시작하는 것은 import 하지 않음
- import module1 as m1 을 사용하면 m1.add(a, b)로 사용 가능하다

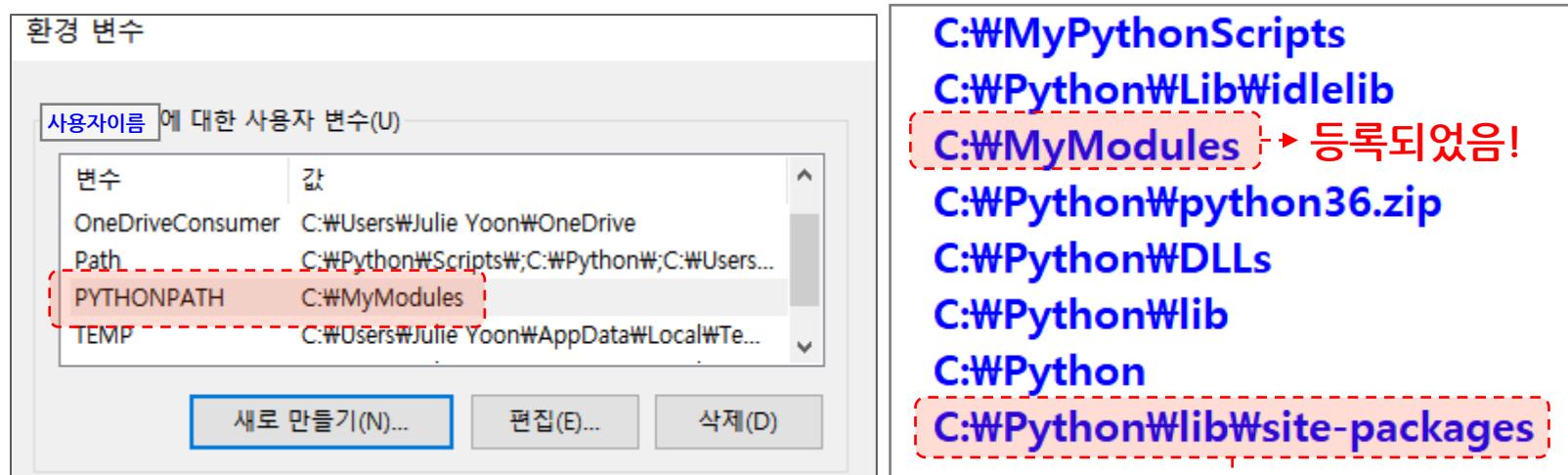
모듈 위치 등록 - PYTHONPATH



▶ 운영체제의 환경 변수에 PYTHONPATH를 추가하고 다시 확인해보자

▶ windows에서 '환경 변수' PYTHONPATH를 사용자 변수로 등록하였다

- 환경 변수 등록 후, 실행 중인 IDLE를 재실행 해야 한다
- 앞에서 작성한 sys.path 확인 코드를 실행하여 목록을 확인한다



- 파이썬 기본 라이브러리 패키지 외에 **추가적인 패키지를 설치하는 디렉터리**
- 사용자 패키지 및 모듈을 이 디렉터리에 위치시키면 모든 경로에서 접근 할 수 있음
- 온라인 패키지 설치가 불가능한 경우 설치된 패키지를 복사해 이곳에 넣어 사용함

모듈의 위치



- python은 모듈을 찾기 위해 다음의 순서로 경로 탐색을 한다

현재 디렉터리 → 파이썬 built-in 모듈 → sys.path에 정의된 디렉터리

- sys.path에 정의되어 있는 디렉터리는 다음과 같다

- 다음의 디렉터리 정보가 문자열 list의 형태로 저장되어 있다

- 현재 작업 디렉터리
- PYTHONPATH 환경변수에 정의되어 있는 디렉터리
- 파이썬과 함께 설치된 기본 라이브러리

```
import sys  
for path in sys.path:  
    print(path)
```

for문으로 아이템 출력

C:/MyPythonScripts
C:\Python\Lib\idlelib
C:\Python\python36.zip
C:\Python\DLLs
C:\Python\Lib
C:\Python
C:\Python\Lib\site-packages

현재 작업 디렉터리

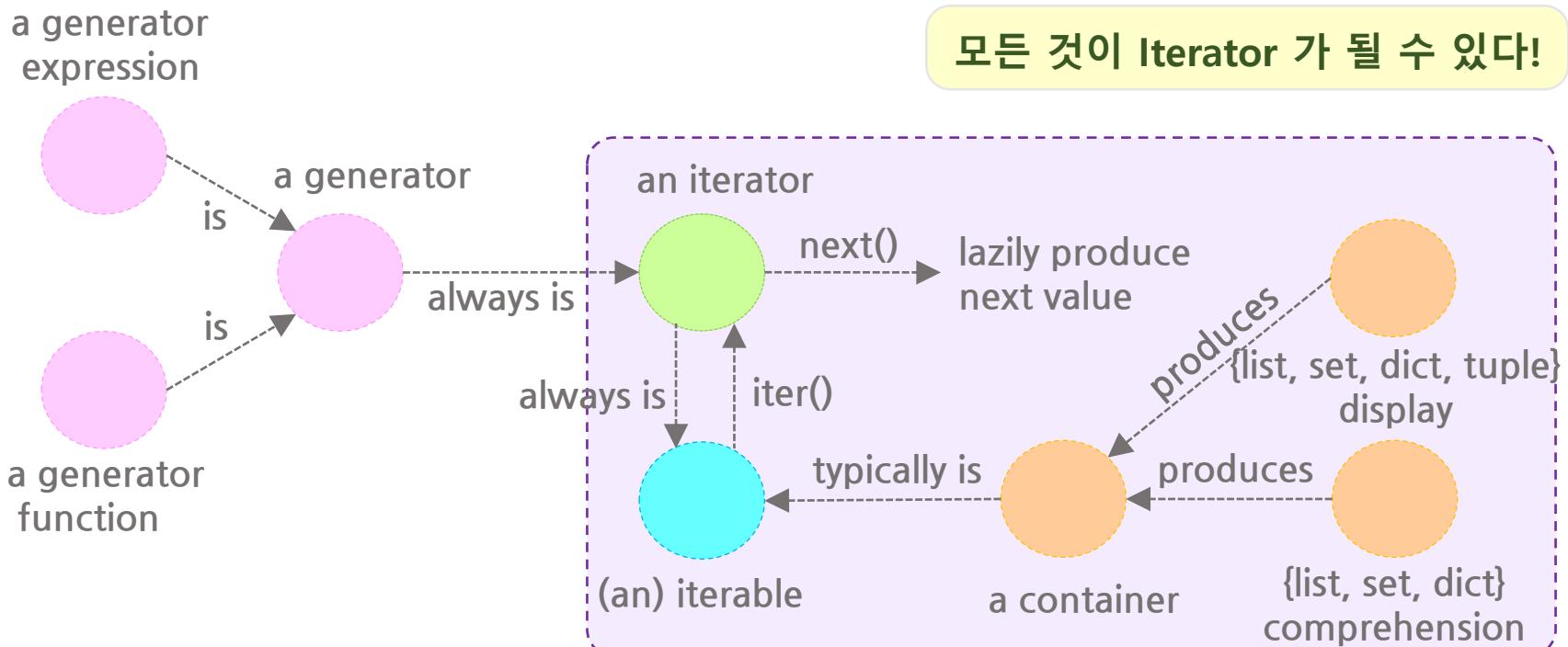
파이썬과 함께 설치된
기본 라이브러리

개념 이해



▣ 다음의 용어들의 관계를 그림으로 이해하여 보자

Iterator, Container, Iterable, Generator, Generator expression, {list, set, dict} comprehension



map, filter, zip 함수



▶ map, filter, zip은 iterable에 대해 동작하는 함수이다

함수	기능
map(function, *iterable)	<ul style="list-style-type: none">▪ 각 항에 function이 적용되어 map 타입 객체로 반환▪ function의 결과가 map의 아이템이 됨▪ 아이템 개수는 변경 없음, 아이템 자체의 변경을 하는 것
filter(function, iterable)	<ul style="list-style-type: none">▪ 각 항에 function을 적용▪ 결과가 True인 아이템 목록을 filter 타입 객체로 반환▪ 아이템 개수가 달라질 수 있음, 아이템 자체 변경 없음
zip(*iterable)	<ul style="list-style-type: none">▪ 동일 개수로 이루어진 iterable을 tuple 쌍으로 묶음▪ zip 타입 객체로 반환

map, filter, zip
“ iterator 객체 ”

iterator 객체의 특징

- for문 및 next() 를 이용해 객체(object) 반환 받고 다음으로 이동
- next()로 한 번 수행되면 앞쪽으로 이동 불가
- print(객체)를 사용하면 객체 타입 및 주소가 출력됨
- list, tuple, dict 의 생성자를 사용해 iterable로 변환 가능

function : map



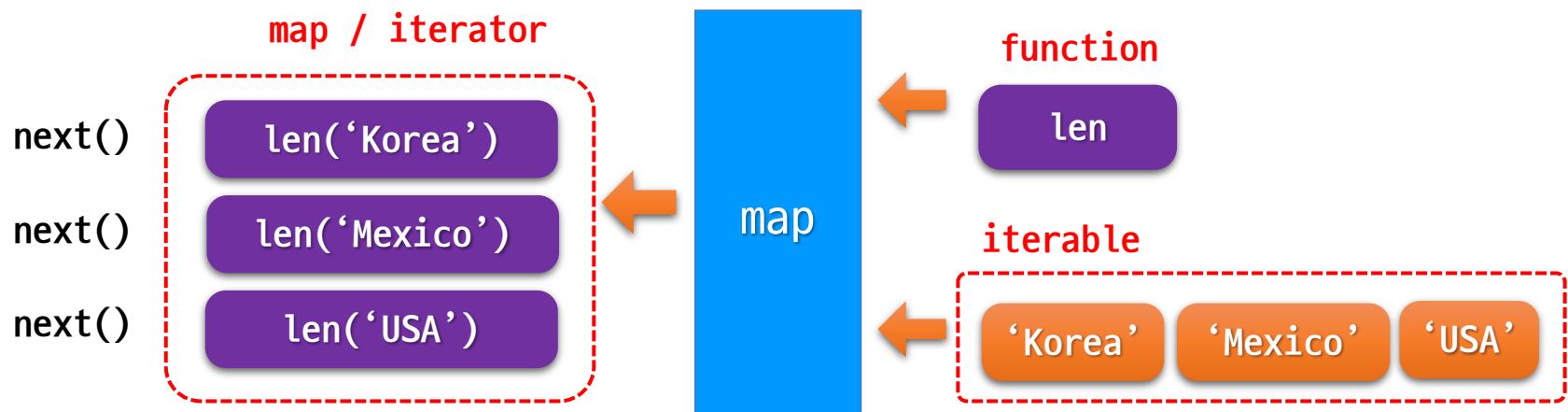
• `map(function, *iterable)` → `map / iterator`

iterable의 각 item에 function이 적용되어 map 객체 반환

function의 parameter 개수 == iterable의 개수 (NOT item의 개수)

next()할 때마다, function의 동작 결과가 반환됨

iterable의 item 개수 만큼 next()를 사용할 수 있음



function : filter



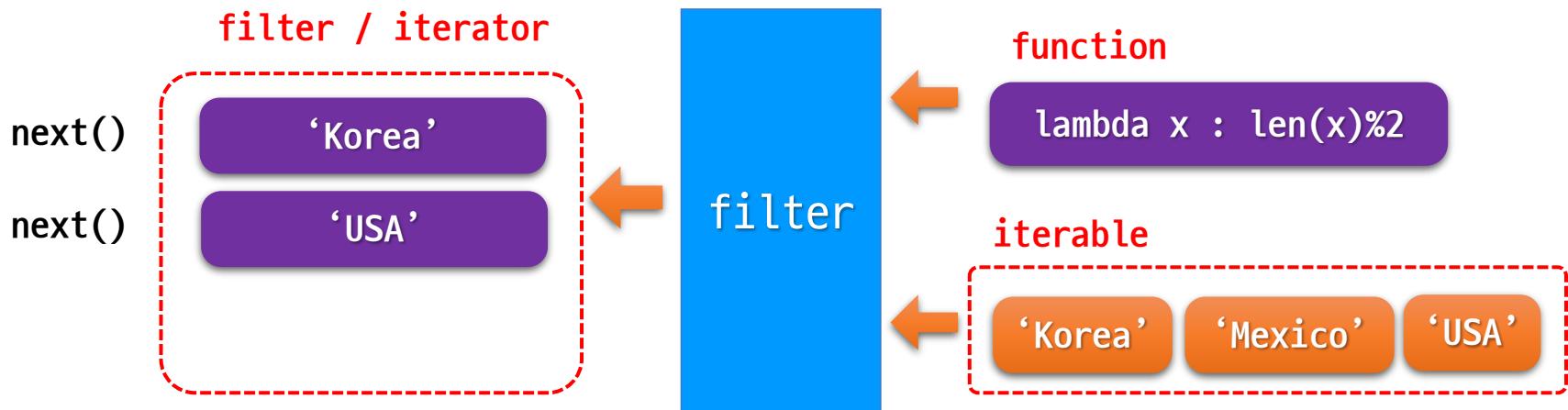
• `filter(function, iterable)`

-> `filter / iterator`

iterable의 각 item에 function이 적용되어 filter 객체 반환

next()할 때마다, function의 결과가 **True인 item**을 반환

iterable의 item 개수 보다 next() 사용 횟수가 적거나 같을 수 있음



function : zip



zip(*function)

-> zip / iterator

동일 item 개수로 이루어진 iterable을 대상으로 함

next()할 때마다, 각 iterable의 동일 위치 item들을 모아 tuple로 반환

iterable의 item 개수 만큼 next()를 사용할 수 있음

filter / iterator

next()

(82, 'Korea')

next()

(52, 'Mexico')

next()

(1, 'USA')

zip

iterable

82

52

1

iterable

'Korea'

'Mexico'

'USA'

function : enumerate

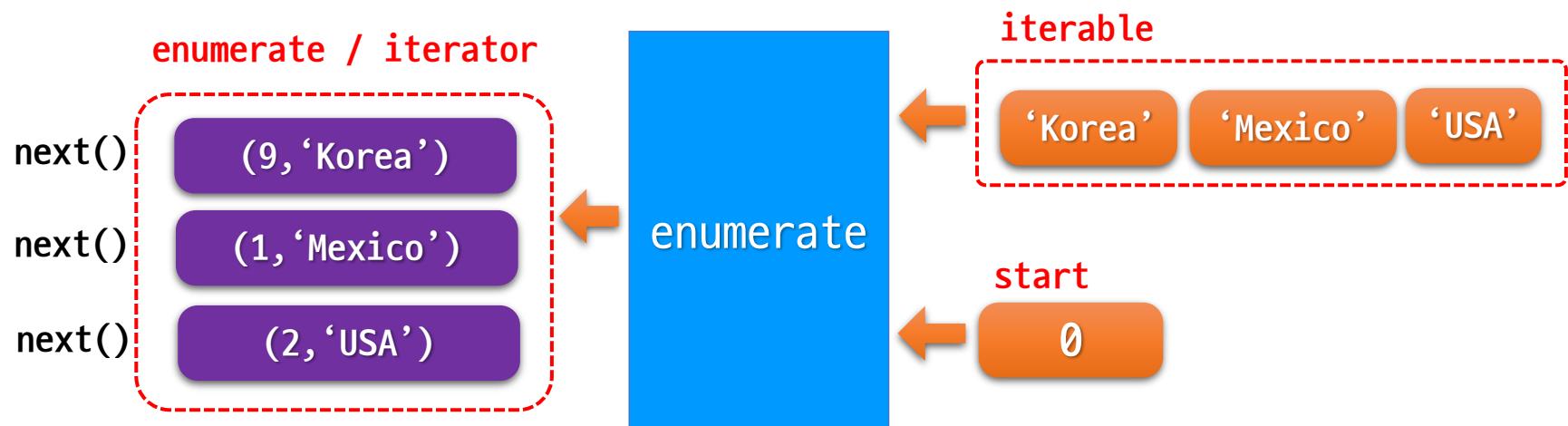


- `enumerate(iterable, start=0)` → `enumerate / iterator`

iterable의 item에 순서 번호가 함께 필요할 때 사용함

`next()`할 때마다, (index, item)의 **tuple**을 반환

iterable의 item 개수 만큼 `next()`를 사용할 수 있음



iterable, iterator, generator



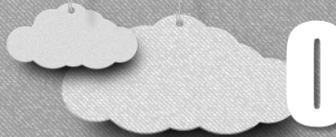
▣ 다음 코드를 통해 iterable, iterator, generator를 비교하여 보자

```
from collections.abc import Iterable, Iterator, Generator  
import sys
```

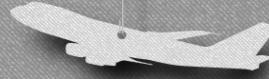
```
r1 = tuple(range(1, 4))  
r2 = tuple(range(1, 1000))  
r3 = range(1, 4)  
r4 = range(1, 1000)  
r5 = myRange(1, 4)  
r6 = myRange(1, 1000)  
r7 = (x for x in range(1, 4))  
r8 = (x for x in range(1, 1000))  
  
for x in r1, r2, r3, r4, r5, r6, r7, r8:  
    print(f'{sys.getsizeof(x):4}',  
          isinstance(x, Iterable),  
          isinstance(x, Iterator),  
          isinstance(x, Generator))
```

객체	종류
r1, r2, r3, r4	Iterable
r5, r6	Iterator
r7, r8	Generator

```
40 True False False  
4024 True False False  
24 True False False  
24 True False False  
32 True True False  
32 True True False  
64 True True True  
64 True True True
```



06



numpy의 ndarray랑 친해져요!



numpy의 이해



- 통계, 분석, AI 분야의 라이브러리 내부에 다양한 수치연산이 필요함
 - 수치연산을 얼마나 효율적으로 처리하는가에 따라 성능에 많은 영향을 줌
 - numpy는 ndarray라는 자료를 바탕으로 강력한 연산 기능을 제공함
-
- numpy와 다른 python 패키지와의 관계



numpy는 다양한 python 패키지에서 데이터 표현과 연산 기능 구현에 사용됨

numpy 설치



- ▶ numpy는 외부 라이브러리이다
- ▶ 표준 파이썬에 포함되지 않으므로 numpy는 import 하여 사용한다
 - np라는 이름으로 numpy 라이브러리를 가져오라는 명령

```
import numpy as np
```

- ▶ numpy 모듈이 설치 되지 않았다면 다음 명령으로 설치한다
 - pip install numpy
 - Windows 의 경우 명령 프롬프트에서 위의 명령을 실행하면 설치된다

- ▶ 자세한 메서드와 속성에 대한 설명은 다음 링크 참조한다
 - <https://docs.scipy.org/doc/numpy/reference/index.html>
 - <https://numpy.org/devdocs/genindex.html>
 - 자세한 parameter 사용에 대한 종류 및 설명을 확인할 수 있다

Vectorizing 연산



▶ numpy는 Vectorizing 연산을 사용한다

▶ vectorizing 연산의 특징 및 장점

- Loop을 사용하지 않고 배열(ndarray)의 각 요소를 계산 함
- 코드의 높은 가독성, 처리 속도 향상
- 대용량 데이터를 다룰 때 사용하면 성능에 큰 도움이 됨

▶ numpy Vectorizing 연산 원리

Python with Loop

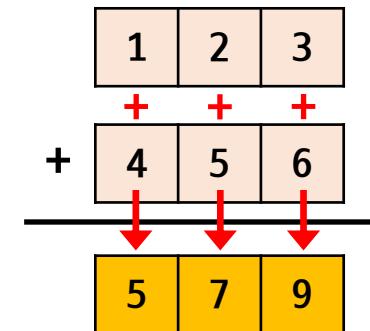
```
A = [ 1, 2, 3]
B = [ 4, 5, 6]

C = []
for a, b in zip(A, B):
    C.append(a + b)
```

python with NumPy

```
import numpy as np
A = np.array([1, 2, 3])
B = np.array([4, 5, 6])
C = A + B
```

numpy 연산 과정
(element by element)



numpy의 성능 예제



- ▶ [예제1] 다음을 실행하여 numpy와 일반 list의 성능 차이를 확인하라
- ▶ 다음은 열의 위치를 변경하는 코드이다

```
a = [[ 1, 2, 3, 4 ] for _ in range(1000000)]  
b = np.array(a)  
print(a[:4], b[:4], sep='\n')
```

데이터 범위를 변경해 가며
차이를 확인해 본다

0:00:00.376637

```
start = datetime.datetime.now()  
c = []  
for x in a:  
    x[2], x[3], x[0], x[1] = x  
    c.append(x)  
end = datetime.datetime.now()  
print(end-start)
```

0:00:00.011031

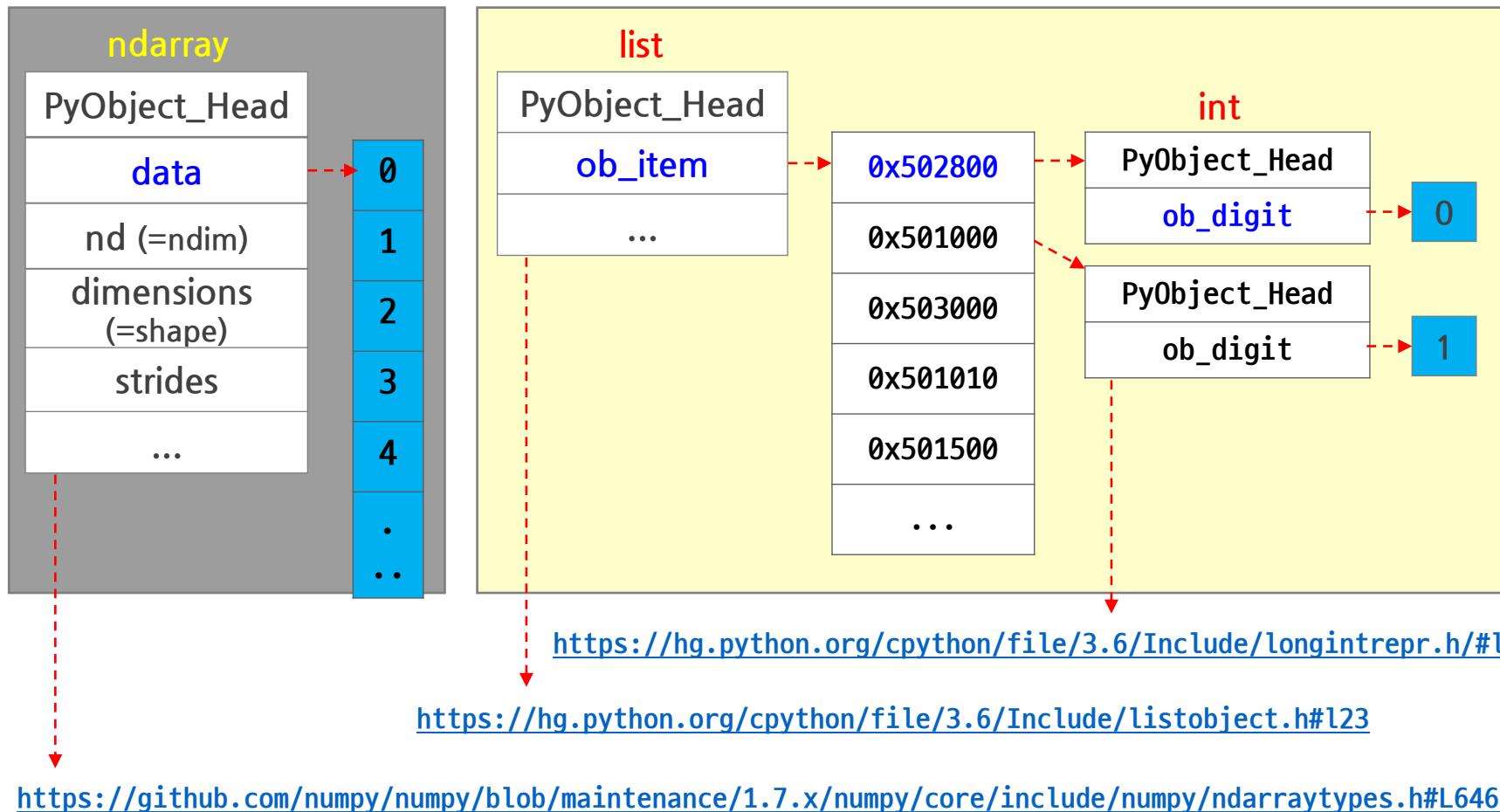
```
start = datetime.datetime.now()  
d = b[ :, [2, 3, 0, 1]]  
end = datetime.datetime.now()  
print(end-start)
```

numpy 를 사용한 코드가 더 간결하며, 빠른 성능인 것을 확인 할 수 있다

ndarray 가 list 보다 빠른 이유는?



▶ numpy의 ndarray와 built-in list의 데이터 접근 방법 차이를 살펴보자



ndarray의 주요 속성



[예제2] ndarray의 주요 속성을 살펴보자

속성	설명
a.ndim	▪ 배열의 깊이(차원)
a.shape	▪ 생성된 배열의 구조에 대한 정보 ▪ tuple 형태, 각 차원 별 요소 개수 정보를 가지고 있음
a.size	▪ 배열의 item 개수
a.dtype	▪ 생성된 배열의 데이터 타입(자료형)에 대한 정보
a.itemsize	▪ item의 메모리 사용 byte 수
a.strides	▪ 메모리에서 데이터의 간격에 대한 정보, shape 및 dtype과 관련이 있음 ▪ tuple 형태, 각 차원 별 데이터 간격 정보를 가지고 있음

```
a = np.array([[0,1,2],[3,4,5]], dtype=np.int32)
```

0	1	2
3	4	5

```
a.ndim      2  
a.shape     (2, 3)  
a.size      6  
a.dtype      int32  
a.itemsize   4  
a.strides   (12, 4)
```

numpy의 자료형



▶ numpy의 자료형은 다음과 같이 분류할 수 있다

자료형	종류
부호 있는 정수형 (int)	int8, int16, int32, int64
부호 없는 정수형 (uint)	uint8, uint16, uint32, uint64
실수형 (float)	float16, float32, float64, float128
복소수(complex)	complex64, complex128, complex256
부울형(boolean)	bool (8bit)
문자열	bytes_(byte string), str_ (unicode string)
객체	object, Container 자료형들은 object로 취급함

numpy 자료형 클래스 사용방법

1. np.int8, np.float32 와 같이 ‘np.자료형’으로 사용
2. 자료형 문자 코드사용 (다음 페이지 참조)
3. 각 자료형 뒤의 8, 16, 32 등의 숫자는 bit 수를 의미하며 8 bit == 1Byte 이다

ndarray 학습 개요



본 과정에서 다루는 numpy의 ndarray 관련 내용은 다음과 같다

ndarray 생성 함수	array-like, 범위(start, stop), random 값 및 0, 1, 특정 값을 사용하여 ndarray를 생성하는 방법
ndarray 변환 함수	ndarray의 shape 또는 dtype을 변경하는 방법
ndarray 연산	산술연산, 비교연산, element-wise, broadcasting
ndarray indexing	다양한 indexer의 사용 방법
numpy의 유용한 함수	unique, where, sum, mean, std, any, all 함수 사용법
ndarray 파일 입출력	ndarray를 파일로 저장하고 불러오는 방법
np.inf, np.nan	np.inf, np.nan에 대한 특징, 의미 학습

ndarray 생성 함수



▶ [예제3] np.array는 array_like를 사용하여 ndarray를 생성하는 함수이다

np.array(array_like, [dtype])

array_like 객체를 인수로 받아 배열 생성

[[1,2,3],[4,5,6]] → array_like, [[1,2,3], [4, 5]] → array_like 아님

Structured Array

- np.array(array_like, dtype) 을 사용하며, array_like를 여러 데이터의 tuple 리스트로 만듦
- 이 경우 Structured Datatype을 작성하여 dtype으로 지정함
- Structured Datatype은 데이터 tuple의 각 필드에 대해 이름과 자료형을 지정하여 작성함
(다양한 작성 방법이 있으며, <https://numpy.org/devdocs/user/basics.rec.html> 를 참조함)
- arr[필드이름] 또는 arr[index] 를 사용하여 indexing 할 수 있음

```
sdtype = [('name', 'U10'), ('height', '<i4'), ('weight', np.float32)]
value = [('Tom', 178, 98.5), ('Jim', 183, 79.5), ('Adam', 175, 82.8)]
a = np.array(value, dtype=sdtype)
```

10글자 문자열(1글자 4byte)

Little Endian, int, 4byte 의미 ('>' → Big Endian)

name	height	weight
Tom	178	98.5
Jim	183	79.5
Adam	175	82.8

ndarray 생성 예 1



[예제3]

```
np.array([[1, 2, 3], [4, 5, 6]])
```

```
[[1 2 3]  
 [4 5 6]]
```

```
np.array([[1, 2, 3], [4, 5]])
```

```
object [list([1, 2, 3]) list([4, 5])]
```

```
sdtype = [('name', 'U10'), ('height', 'i4'), ('weight', np.float32)]  
value = [('Tom', 178, 98.5), ('Jim', 183, 79.5), ('Adam', 175, 82.8)]  
a = np.array(value, dtype=sdtype)
```

Structured Array

```
a
```

```
[('Tom', 178, 98.5) ('Jim', 183, 79.5) ('Adam', 175, 82.8)]
```

```
a.dtype
```

```
[('name', 'U10'), ('height', 'i4'), ('weight', 'f4')]
```

```
a['name']
```

```
['Tom' 'Jim' 'Adam']
```

```
a[1]
```

```
('Jim', 183, 79.5)
```

name	height	weight
Tom	178	98.5
Jim	183	79.5
Adam	175	82.8

ndarray 생성 함수 2/4



▶ [예제3] 값 범위(start, stop)을 사용하여 ndarray를 생성하는 함수들이다

- `np.arange([start,] stop [, step,] dtype=None)`

- range와 같은 원리로 배열 생성

- `np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)`

- start부터 stop까지 동일 step의 데이터를 num개 갖는 배열 생성

- endpoint : stop까지 포함할 경우 True 포함하지 않을 경우 False

- retstep : True인 경우 ndarray, step의 tuple을 반환

```
a = np.arange(5)      [0 1 2 3 4]  
b = np.arange(1, 10, 2) [1 3 5 7 9]
```

```
a = np.linspace(1, 5, 9)  
b = np.linspace(1, 5, 10, endpoint=False)  
c = np.linspace(1, 5, 9, retstep=True)
```

```
[1. 1.5 2. 2.5 3. 3.5 4. 4.5 5. ]  
[1. 1.4 1.8 2.2 2.6 3. 3.4 3.8 4.2 4.6]  
(array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ]), 0.5)
```

linear space

ndarray 생성 함수 3/4



▶ [예제4] random을 사용하여 ndarray를 생성하는 함수들이다

함수	기능
<code>np.random.rand(d0, d1, ...dn)</code>	<ul style="list-style-type: none">▪ 0 ~ 1 사이의 균일 분포로 실수 난수 배열 생성▪ d0 ... dn 은 배열의 shape를 의미하는 정수
<code>np.random.randint(e, size=n)</code> <code>np.random.randint(s, e, size=n)</code>	<ul style="list-style-type: none">▪ 0 ~ e-1 또는 s ~ e-1 값을 갖는 균일 분포의 정수 난수 생성▪ size : 정수(1차원) 또는 tuple(2차원 이상)로 shape 지정
<code>np.random.randn(d0, d1, ...dn)</code>	<ul style="list-style-type: none">▪ 평균 0, 표준편차 1을 갖는 가우시안 표준 정규 분포로 난수 배열 생성 (음수도 포함됨)
<code>np.random.normal(평균, 표준편차, size)</code>	<ul style="list-style-type: none">▪ 평균과 표준편차를 지정하여 size 개의 난수로 배열 생성▪ randn()과 normal()은 정규 분포를 갖음
<code>np.random.permutation(e)</code> <code>np.random.permutation(c)</code>	<ul style="list-style-type: none">▪ e 는 정수, c 는 array_like 객체▪ 0 ~ e-1 값을 갖는 정수의 무작위 순서 1차원 배열 생성▪ c 요소들에 대해 무작위 순서로 변경된 1차원 배열 생성▪ c 가 다차원인 경우 가장 상위 차원만 무작위 순서로 변경 됨

random 참조 : <https://docs.scipy.org/doc/numpy/reference/routines.random.html>

ndarray 생성 예 2



```
np.random.rand(5)
```

```
[0.9088 0.8697 0.536 0.7954 0.2754]
```

```
np.random.randint(50, 100, (2,4))
```

```
[[95 74 65 93]
 [53 70 74 61]]
```

```
np.random.randn(200, 50)
```

```
[[ -0.9049 -1.11 ... -0.168 -0.6898]
 ...
 [-1.1395 0.0986 ... 0.9346 -1.0711]]
 mean : 0.013, std : 0.993
```

```
np.random.normal(2, 3, (100, 20))
```

```
[[ 1.6929 2.1374 ... -0.6581 0.5421]
 ...
 [ 2.7684 1.4454 ... 2.5391 -1.1127]]
 mean : 2.030, std : 3.040
```

```
np.random.permutation(10)
```

```
[1 2 6 9 0 4 3 5 7 8]
```

```
data = [1,3,4,5,3,1,3]
np.random.permutation(data)
```

```
[5 1 1 3 4 3 3]
```

np.set_printoptions 를 사용한 배열 출력 옵션 설정 (보기 좋은 배열 출력)

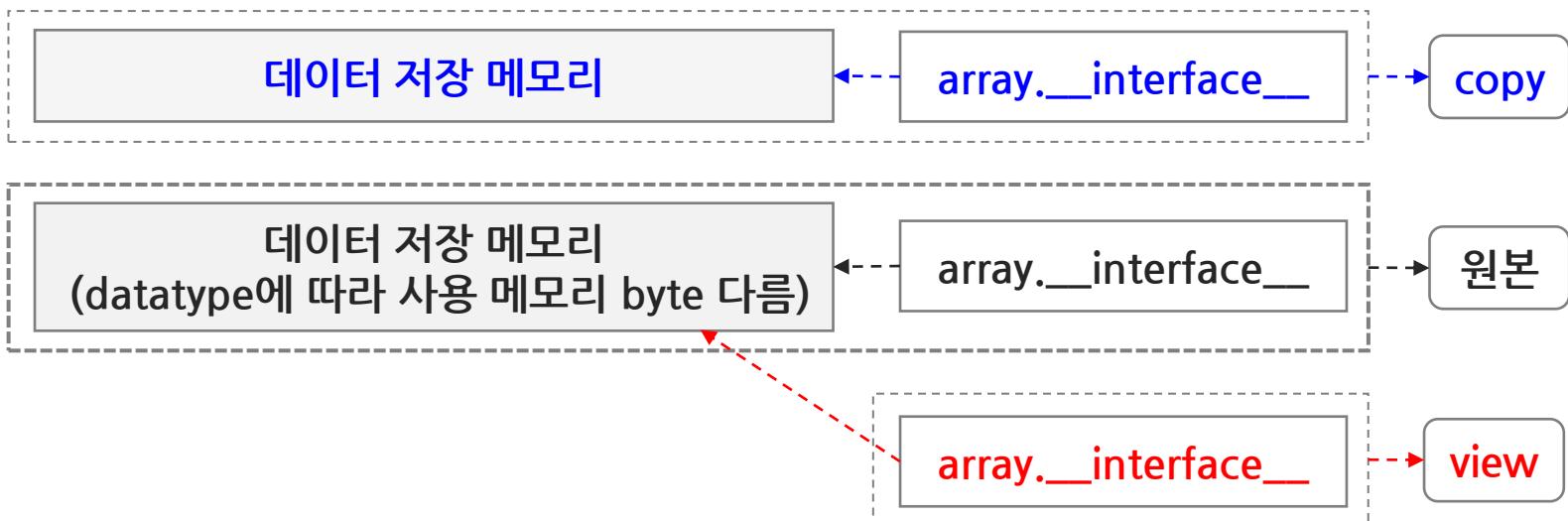
https://numpy.org/doc/stable/reference/generated/numpy.set_printoptions.html

copy & view 객체



▶ [예제5] ndarray의 copy 객체 또는 view 객체의 특징을 살펴보다

copy 객체	view 객체
<ul style="list-style-type: none">▪ arr.copy()를 사용하여 생성▪ 원본과 다른 메모리에 데이터 복사	<ul style="list-style-type: none">▪ arr.view()를 사용하여 생성▪ 원본 배열과 같은 데이터 참조



ndarray 배열 변환/조작 함수 -1/3



▶ [예제6] ndarray 를 다른 속성의 ndarray로 변환/조작하는 함수들이다

변환 함수	동작
<code>a.reshape(shape, order='C')</code> <code>np.reshape(a, shape, order='C')</code>	<ul style="list-style-type: none">▪ 배열의 ‘shape’을 변환하여 반환함▪ arr와 shape의 size는 동일해야 함▪ shape의 요소 중 1개를 -1로 하여 자동 계산 가능▪ order='F'로 지정하면 copy 객체가 반환 됨
<code>a.astype(dtype, order='K')</code>	<ul style="list-style-type: none">▪ 배열의 데이터 ‘타입’을 변환하여 반환함

```
a = np.array([[1,2,3],[4,5,6]], dtype=np.int32)
b = a.astype(np.int64)
c = a.flatten()
for n, x in zip("abc", (a,b,c)):
    npinfo(n, a, x)

name      : a
share     : True
data      : [[1, 2, 3], [4, 5, 6]]
shape     : (2, 3)
stride   : (12, 4)

name      : b
share     : False
data      : [[1, 2, 3], [4, 5, 6]]
shape     : (2, 3)
stride   : (24, 8)

name      : c
share     : False
data      : [1, 2, 3, 4, 5, 6]
shape     : (6,)
stride   : (4,)
```

배열의 shape 변경 - reshape



▶ [예제6] reshape을 사용하면 배열의 shape을 변경할 수 있다

```
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(1, -1)
c = a.reshape(-1)
d = a.reshape(-1, 1)
e = a.reshape(-1, 1, order='F')
```

```
np.may_share_memory(a, x)
→ 두 배열이 view 관계인 경우 True 반환
x.tolist()
→ ndarray x를 list 객체로 변환하여 반환
```

```
name : b
share : True
data : [[1, 2, 3, 4, 5, 6]]
shape : (1, 6)
```

```
name : d
share : True
data : [[1], [2], [3], [4], [5], [6]]
shape : (6, 1)
```

```
name : c
share : True
data : [1, 2, 3, 4, 5, 6]
shape : (6, )
```

```
name : e
share : False
data : [[1], [4], [2], [5], [3], [6]]
shape : (6, 1)
```

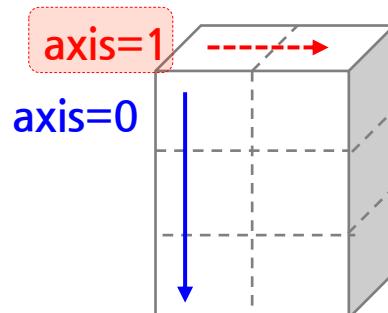
axes, axis의 이해



- ndarray에 사용되는 axes 또는 axis 는 ‘축’을 의미한다
- shape 표기 법에 따라 앞에 표기되는 축이 0번 그 뒤로 1씩 증가한 번호가 된다
 - 축 번호를 음수로 사용할 수 있으며, 뒤에서부터 -1, 그 앞으로 1씩 감소한 번호가 된다

0	1
2	3
4	5

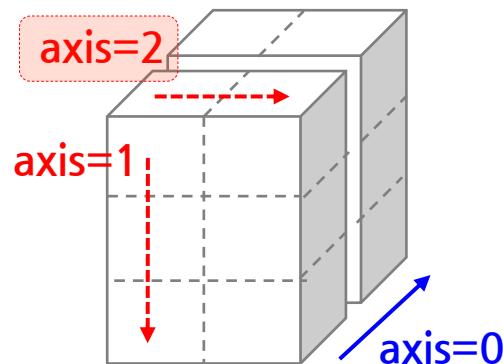
ndim : 2, shape : (3, 2)



axis = -1 과 동일

0	1	6	7
2	3	8	9
4	5	10	11

ndim : 3, shape : (2, 3, 2)



numpy의 산술 연산



[예제7] numpy의 산술 연산을 살펴보자

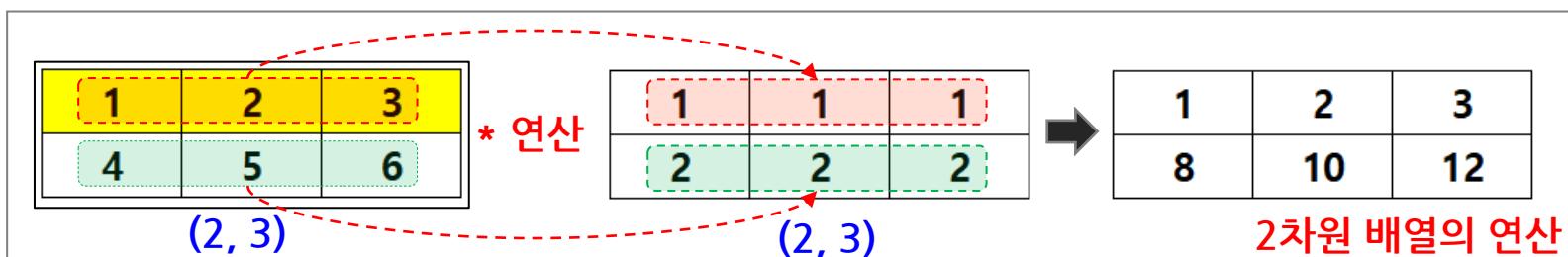
연산자	연산	결과
+	$a + b$	[3 9 17]
-	$b - a$	[1 3 7]
*	$a * b$	[2 18 60]
/	b / a	[2. 2. 2.4]

- `a = np.array([1, 3, 5])`
- `b = np.array([2, 6, 12])`

`**, //, %` 연산도 가능하다

element-wise

- 같은 shape을 갖는 두 배열 간의 연산을 말하며, numpy 연산의 기본이다
- 같은 차원의 배열에서 각 차원을 구성하는 요소 수가 다르면 오류가 발생한다



broadcasting의 이해 -1/2



- ▶ numpy의 배열 연산은 element-wise 방식으로 이루어진다
- ▶ 다른 차원의 두 배열 연산 시 broadcasting을 통해 배열 전환 후 연산이 수행된다
- ▶ broadcasting의 배열 전환 원리는 다음과 같다
 - 한 배열 shape이 다른 배열 shape의 부분 shape이면 차원이 높은 배열 shape을 따른다
 - $(2, 3, 4) + (3, 4) \rightarrow (2, 3, 4)$ $(4, 2) + (2,) \rightarrow (4, 2)$ $(2, 4) + (3, 1, 2, 4) \rightarrow (3, 1, 2, 4)$
 - 요소 개수가 1인 차원에 대해서는 다른 배열의 동일 차원을 요소 수를 따른다
 - $(5, 1) + (3,) \rightarrow (5, 3)$ $(5, 1, 4) + (3, 1) \rightarrow (5, 3, 4)$ $(4, 2, 1) + (1, 3) \rightarrow (4, 2, 3)$
 - 스칼라 값은 상대 배열의 shape과 같은 shape, 동일 값으로 구성된 배열이 된다
 - $(2, 3) + 2 \rightarrow (2, 3)$

1	2	3
4	5	6

A 배열 : (2, 3)

2



스칼라 값 : 2

broadcasting

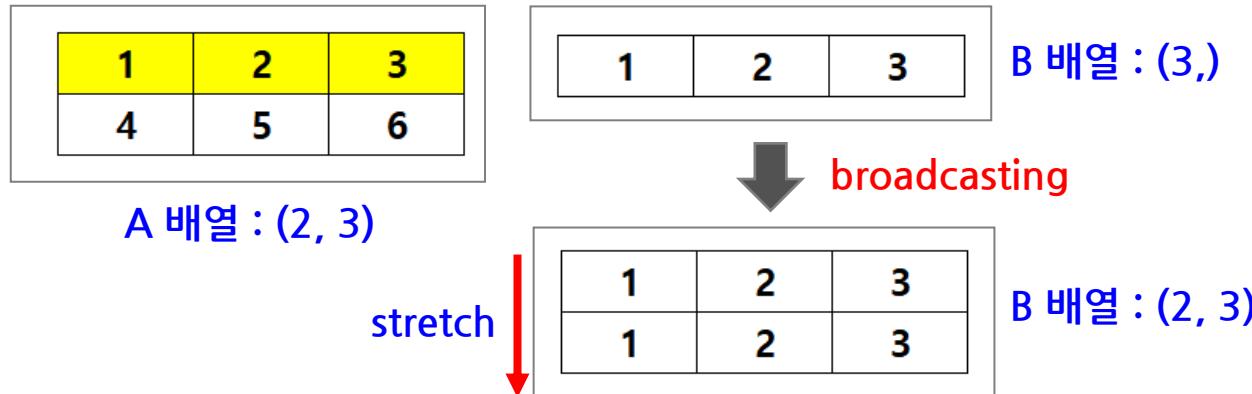
2	2	2
2	2	2

B 배열 : (2, 3)

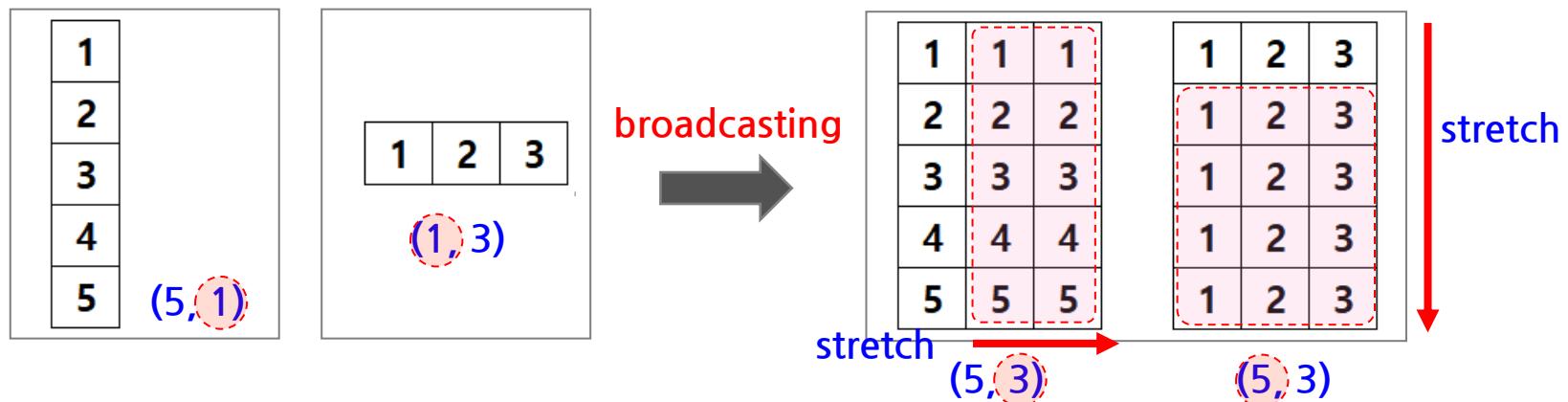
broadcasting의 이해 -2/2



- 두 개 배열의 shape이 같도록 broadcast 하여 연산한다
- 부분 배열인 경우는 큰 쪽에 맞추어 확장 된다



- 요소 개수가 1인 axis는 상대 배열의 동일 axis의 요소 개수만큼 확장 된다



numpy의 broadcasting 연산



▶ [예제8] 배열 연산 결과를 분석하여 broadcasting을 이해하여 보자

```
a=np.array([[1, 1, 1], [1, 1, 1]])      c=np.array([[1],[2],[3]])
b=np.array([[1, 1, 1], [2, 2, 2]])      d=np.array([3, 3, 3])
```

종류	차원 계산	연산	결과
element-wise	(2, 3) + (2, 3) : (2, 3)	a + b	[[2 2 2] [3 3 3]]
broadcasting	(2, 3) + (3,) : (2, 3)	a + d	[[4 4 4] [4 4 4]]
	(2, 3) + 스칼라 : (2, 3)	b + 4	[[5 5 5] [6 6 6]]
	(3, 1) + (3,) : (3, 3)	c + d	[[4 4 4] [5 5 5] [6 6 6]]



A 배열 : (3,)

B 배열 : (2,)

ndarray의 비교 연산



▶ [예제9] 코드를 실행하여 비교 연산을 이해하여 보자

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

```
b=np.array([[1, 3, 5], [2, 4, 6]])
```

```
c=np.array([4, 3, 3])
```

차원 계산	연산	결과
(2, 3) == (2, 3) : (2, 3)	a == b	[[True False False] [False False True]]
(2, 3) != (2, 3) : (2, 3)	a != b	[[False True True] [True True False]]
(2, 3) > (2, 3) : (2, 3)	a > b	[[False False False] [True True False]]
(2, 3) < (2, 3) : (2, 3)	a < b	[[False True True] [False False False]]
(2, 3) >= (3,) : (2, 3)	a >= c	[[False False True] [True True True]]
(2, 3) <= 스칼라 : (2, 3)	b <= 3	[[True True False] [True False False]]

indexing



▶ [예제10] indexing을 사용하여 배열에서 원하는 데이터를 참조/변경한다

▶ 배열 이름 뒤에 [] 를 사용하여 참조/변경 할 데이터를 표기한다

- 결과는 원본의 view이며, view에 대입을 통해 데이터 변경이 가능함
- ndim >= 2 인 경우 [] 내부에 콤마(,)를 사용하여 차원(축, axes, axis)을 구분함
- 콤마(,) 사이에는 참조/변경하기 원하는 데이터에 대한 표기(indexer)가 포함되어야 함
- 다양한 indexer가 존재함
- axis=0을 제외한 차원에 대한 표기 생략가능, 생략은 “모두 선택”의 의미를 갖음

```
a = np.arange(9).reshape(3,3)  
  
print(a)  
print(a[0, 2])  
print(a[1])  
print(a[2][1])
```

```
[[0 1 2]  
 [3 4 5]  
 [6 7 8]]  
2  
[3 4 5]  
7
```

- <https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html>
- <https://www.numpy.org/devdocs/user/basics.indexing.html>

indexer의 종류



▶ [예제10] ndarray에서 사용할 수 있는 indexer의 종류는 다음과 같다

indexer	설명
single element index	▪ integer object
slice and stride	▪ slice object ➔ start:stop:step
index arrays	▪ sequence-like, numpy object
Boolean arrays	▪ boolean array : the same shape with original array
structural indexing tools	▪ ..., np.newaxis
index + slice	▪ combine index and slice

```
a = np.arange(5)

print(a[1])
print(a[1:4:2])
print(a[[0,1,-1]])
print(a[[True, False, False, True, False]])
print(a[:, np.newaxis])
```

```
1
[1 3]
[0 1 4]
[0 3]
[[0]
 [1]
 [2]
 [3]
 [4]]
```

Indexing의 반환 값



- ▶ [예제11] 아래의 코드 실행 결과를 예측한 뒤 실행하여 결과를 확인하라
- ▶ ndarray의 indexing의 반환 값은 원본을 참조하는 view이다
 - 새로운 ndarray로 처리하려면 copy 함수나 copy 메소드를 사용해야 한다

```
# list of python
x = [[1, 2, 3], [6, 7, 8]]
z = x[:2]
z[0] = [10, 20, 30]
print(x, z, sep='\n')
print("-"*25)

# ndarray of numpy
x = np.array([[1, 2, 3], [6, 7, 8]])
z = x[:2]
z[0] = [10, 20, 30]
print(x, z, sep='\n')
print(np.may_share_memory(x,z))
```

```
[[1, 2, 3], [6, 7, 8]]
[[10, 20, 30], [6, 7, 8]]
-----
[[10 20 30]
 [ 6  7  8]]
[[10 20 30]
 [ 6  7  8]]
True
```

→ x, z의 원본 데이터가 같은 객체인지 반환
True인 경우 원본 데이터가 같은 객체임

Indexing - single element index



▶ [예제12] single element index를 사용한 indexing을 살펴보자

방법	설명
a[n] (1D array)	<ul style="list-style-type: none">n번 index 요소를 반환 (index는 0부터 시작하는 일련번호, 음수가 능)
a[n, m, ...] (N D array)	<ul style="list-style-type: none">axis=0부터 뒤쪽으로 콤마(,)를 기준으로 index를 나열함index 표기되지 않은 차원은 “모두 선택” 예) 2차원 배열에서 a[n]은 n번 행 전체를 반환 함indexing을 연속으로 진행한 것과 콤마(,) 나열은 같은 결과 예) $a[n][m] == a[n, m]$

<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td></tr></table>	0	1	2	3	4	5	<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	8	9	10	11
0	1												
2	3												
4	5												
6	7												
8	9												
10	11												
$c : (2, 3, 2)$													

<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td></tr></table>	0	1	2	3	4	5	<table border="1"><tr><td>8</td><td>9</td></tr></table>	8	9	7	$c[1, 0, 1]$
0	1										
2	3										
4	5										
8	9										
	$c[0]$										

(~~✖~~ 3, 2) \rightarrow (3, 2) (~~✖✖~~ 2) \rightarrow (2,) (~~✖✖✖~~) \rightarrow ()

single element index의 사용이 1개 증가 할 때마다 배열의 차원(ndim)은 1씩 감소한다

Indexing - slice & stride



▶ [예제12] slice & stride를 사용한 indexing을 살펴보자

n, m, k 는 정수이며, 생략 가능, 음수 일 수 있으며, -1은 마지막 요소의 번호

방법	설명
a[n:m:k] (1D array)	<ul style="list-style-type: none">n 부터 m-1 까지, k 씩 건너 뛴 위치 요소로 구성된 배열 반환
a[n1:m1:k1, n2:m2:k2, ...] (N D array)	<ul style="list-style-type: none">axis=0부터 뒤쪽으로 콤마(,)를 기준으로 index를 나열함index 표기되지 않은 차원은 “모두 선택”모든 정수가 생략된 index에 대해서는 “모두 선택”을 적용함<ul style="list-style-type: none">→ 특정 차원에 대해 모든 범위를 의미함→ 나머지 모든 차원에 대해 모든 범위를 의미함

<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td></tr></table>	0	1	2	3	4	5	<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	8	9	10	11
0	1												
2	3												
4	5												
6	7												
8	9												
10	11												
$c : (2, 3, 2)$													

차원(ndim) 변경 없음

<table border="1"><tr><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td></tr><tr><td>10</td><td>11</td></tr></table>	6	7	8	9	10	11	<table border="1"><tr><td>7</td><td>6</td></tr><tr><td>1</td><td>0</td></tr><tr><td>3</td><td>2</td></tr><tr><td>5</td><td>4</td></tr></table>	7	6	1	0	3	2	5	4	<table border="1"><tr><td>10</td><td>11</td></tr><tr><td>4</td><td>5</td></tr><tr><td>2</td><td>3</td></tr><tr><td>0</td><td>1</td></tr></table>	10	11	4	5	2	3	0	1
6	7																							
8	9																							
10	11																							
7	6																							
1	0																							
3	2																							
5	4																							
10	11																							
4	5																							
2	3																							
0	1																							
$c[1:] (1, 3, 2)$	$c[...,-1] (2, 3, 2)$	$c[:, -1] (2, 3, 2)$																						

Indexing - index_array



▶ [예제12] index_array를 사용하여 원하는 항목을 정교하게 표현할 수 있다

방법	설명
a[index_array] (1D array)	<ul style="list-style-type: none">index_array에 포함된 index 항목들로 구성된 배열 반환a[[0, 2, 4]] 는 a에서 0,2,4 번 index 항목으로 구성된 배열 반환index_array 사용시 주의 사항<ul style="list-style-type: none">- index는 중복해서 사용할 수 있음- 2차원 이상의 index_array 는 ndarray를 사용([[[➔ 사용 불가능)
a[idx_a1, idx_a2, ...] (N D array)	<ul style="list-style-type: none">idx_a1, idx_a2의 같은 위치 항들이 묶어 위치 정보가 됨다차원이 되면 콤마(,)로 나열하며 동일 길이로 구성되어야 함a가 2D array인 경우<ul style="list-style-type: none">- a[[0,0,1],[2,3,1]] 은 [0,2], [0,3], [1,1] 항을 배열로 반환- a[[0, 1], 0] : a[[0,1], [0,0]] (broadcasting) [0, 0], [1, 0] 항 반환

x

1	2	3
---	---	---

 x = np.array([1, 2, 3])
single element index ← print(x[1], x[[1]])
print(x[np.array([[0,0], [2,1]])])
print(x[[2, 0, 1, 1]])

2 [2]
[[1 1]]
[3 2]]
[3 1 2 2]

index_array의 형태에 따라 ndim이 동일하거나, 증가, 감소 될 수 있다

Indexing - boolean index 1/2



[예제12] Boolean 배열을 사용한 indexing을 살펴보자

방법	설명
<code>a[boolean_index]</code>	<ul style="list-style-type: none">boolean_index는 배열과 동일한 shape의 Boolean 배열임boolean_index는 ndarray, list 등으로 작성, 연산 결과 일 수 있음주로 결과가 True/False인 비교연산식을 사용하여 작성함복잡한 조건은 np.logical_and(), np.logical_or(), np.logical_not() 활용Boolean 배열과 &, , ~ 연산자를 사용하여 조건작성 가능 (괄호 사용 중요)

a	1	2	3	4	5	6
b	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
a[b]	1	3	4	6		

Boolean 배열

a	1	2	3	4	5	6
<code>a%2==0</code>	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
<code>a[a%2==0]</code>	2	4	6			

대상의 차원에 상관없이
결과 배열은 1차원이다

Indexing - boolean index 2/2



▶ [예제12] 논리형 배열을 기반으로 인덱싱을 할 수 있다

boolean indexing	의미
<code>a[a%2==0]</code>	2의 배수인 값
<code>a[a>=4]</code>	4보다 크거나 같은 값
<code>a[np.logical_or(a%3==0, a%2 == 0)]</code>	3의 배수 또는 2의 배수 인 것
<code>a[np.logical_and(a>=6, a<=9)]</code>	6~9 사이의 값
<code>a[np.logical_not(a%3 == 0)]</code>	3의 배수가 아닌 것
<code>a[a%3==0] = -1</code>	

마지막 동작의 의미를 적어 보도록 한다

Indexing 활용



▶ [예제13] 다음은 indexing 활용에 대한 예시이다

aN3 은 (N,3) shape, aN 은 (N,) shape, a 는 임의 shape의 배열이다

indexing	의미
1 aN3[:, [1, 0, 2]]	0번과 1번 열을 바꿈
2 a3N[[0, 2, 1], :]	1번과 2번 행을 바꿈
3 aN[:, None] == aN[:, np.newaxis]	aN.reshape(-1, 1) 과 같은 결과
4 aN3[aN3[:, 0]<0]	0번째 열의 값이 0보다 작은 행
5 aN3[aN3[:,0]<0, 1:3]	0번째 열의 값이 0보다 작은 행의 1, 2번 열

<code>[[0 1 2]</code>	<code>1</code>	<code>[[0 0 0]</code>	<code>2</code>	<code>[[0 1 2]</code>	<code>3</code>
<code>[0 1 2]</code>		<code>[1 1 1]</code>		<code>[2 2 2]</code>	
<code>[0 1 2]</code>		<code>[2 2 2]</code>		<code>[0 1 2]</code>	
<code>[0 1 2]</code>		<code>[0 0 0]</code>		<code>[1 1 1]</code>	
<code>-----</code>		<code>-----</code>		<code>-----</code>	
<code>[[1 0 2]</code>		<code>[[0 0 0]</code>		<code>[[1 0 0]</code>	
<code>[1 0 2]</code>		<code>[2 2 2]</code>		<code>[-5 2 2]</code>	
<code>[1 0 2]</code>		<code>[1 1 1]</code>		<code>[5 3 3]</code>	
<code>[1 0 2]</code>				<code>-----</code>	

<code>[[-1 0 0]</code>	<code>4</code>	<code>[[-1 0 0]</code>	<code>5</code>
<code>[1 1 1]</code>		<code>[-5 2 2]</code>	
<code>[-5 2 2]</code>		<code>[5 3 3]</code>	
<code>[5 3 3]</code>		<code>-----</code>	
<code>[[0 0]]</code>		<code>[[0 0]]</code>	
<code>[2 2]]</code>		<code>[2 2]]</code>	

numpy의 유용한 함수 - 1



▶ [예제14] numpy의 유용한 함수들은 다음과 같다 (a는 ndarray 객체)

함수	기능
<code>np.mean(a, axis=None)</code>	▪ 배열내 모든 원소의 평균을 스칼라 값으로 반환 (float64)
<code>np.sum(a, axis=None)</code>	▪ 배열내 모든 원소의 합을 스칼라 값으로 반환
<code>np.std(a, axis=None)</code>	▪ 배열내 모든 원소의 표준 편차를 스칼라 값으로 반환
<code>np.any(a, axis=None)</code>	▪ 배열의 원소들 중 참이 하나라도 있으면 True 반환 ▪ 0, False, None → 거짓
<code>np.all(a, axis=None)</code>	▪ 배열의 모든 원소가 참인 경우 True 반환

- `axis=0`은 열, `axis=1`은 행에 대한 연산 결과를 배열로 반환
- ndarray에도 동일 동작을 하는 메서드가 있음 (`arr.mean()`, `arr.sum()` 등)

`np.sum(a, axis = 0)`

1	2	3
4	5	6

[5 7 9]

`np.sum(a, axis = 1)`

1	2	3
4	5	6

[6 15]

numpy의 유용한 함수 - 2



▶ [예제21] 조건에 따른 배열 반환 함수

• np.where(조건, [,c, c])

- 조건 : 참, 거짓(=0, False, None)으로 이루어진 배열 또는 스칼라
- 조건 배열이 참일 때 c, 거짓일 때 d를 취한 배열 반환 (c, d→스칼라 가능)
- c, d 생략 시 조건이 참인 것에 대한 index 배열(ndarray) tuple 반환
(ndarray, ndarray, ...)의 형태, 조건 배열의 ndim == len(tuple)
- 복잡한 조건은 ~, &, |, 및 np.logical_and(), np.logical_or() 사용
→ boolean index 참조

```
a = np.array([4, 8, 2, 5])
b = np.array([3, 9, 1, 7])
print(np.where(True, a, b))
print(np.where(False, a, b))
print(np.where(a<5, a, 10*b))
print(np.where([[True, False], [False, True]], 
              [[3, 4], [5, 7]],
              [[1, 2], [9, 2]]))
```

```
[4 8 2 5]
[3 9 1 7]
[ 4 90  2 70]
[[3 2]
 [9 7]]
```

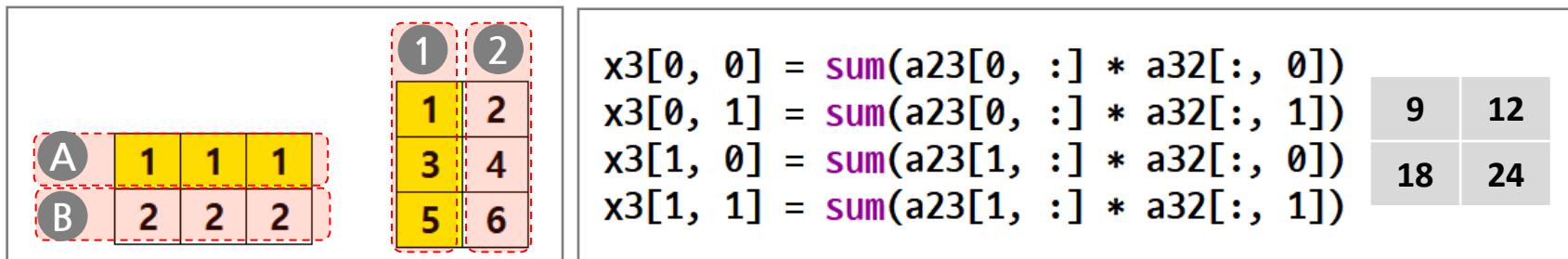
numpy의 유용한 함수 - 2



☞ dot와 matmul의 공통 연산 방법을 알아보자

☞ dot와 matmul은 행렬 연산이므로 “앞 배열의 열, 뒤 배열의 행”수가 같아야 한다

- (2, 3) 앞 배열과 (3, 2) 뒤 배열 연산 결과 (2, 2) shape의 배열이 생성된다



shape : 2, 3

행 위치

shape : 3, 2

열 위치

shape : 2, 2

다음 배열 연산에 대한 그림을 그리고 연산 결과를 예측하여 보자

- A : (2,3), B : (3,2)일 때 np.dot(B, A) 및 np.matmul(B, A)가 가능한가?
- 가능하다면 결과의 shape은 무엇인가?

np.matmul, np.dot 공통



a, b 모두 1-D

[1 2 3]

[1 2 3]

14

a, b 모두 2-D

[[1 2]
[3 4]
[5 6]]

[[1 1]
[2 2]]

[[5 5]
[11 11]
[17 17]]

a 1-D, b N-D

[1 2 3]

[[1]
[2]
[3]]

[14]

[1 2 3]

[[[1 2]
[3 4]
[5 6]]]

[[22 28]
[9 11]]

[[1 1]
[1 2]
[2 2]]]

a N-D, b 1-D

[[1 1 1]
[2 2 2]]

[1 2 3]

[6 12]

[[[1 2 3]
[4 5 6]]
[[1 1 1]
[2 2 2]]]

[1 2 3]

[[14 32]
[6 12]]

dot 연산의 이해



▶ [예제26] 3차원 이상에서 dot(=inner product) 연산 원리는 다음과 같다

▶ 대응하는 열/행을 삭제하고 앞 배열에서 뒤 배열 쪽으로 남은 index를 나열한다

- 앞 배열의 열과 뒤 배열의 행의 수가 같은 경우 연산이 가능하다
- 단, 뒤 배열이 1차원인 경우 배열 크기를 행의 수로 취급한다

- (2, 2, 3)과 (2, 3, 3) → (2, 2, 2, 3)
- (2, 2, 3)과 (3, 3, 2) →
- (2, 2, 3)과 (3, 3) →
- (2, 2, 3)과 (3, 1) →
- (2, 2, 3)과 (3,) →
- (2, 3)과 (2, 3, 2) →
- (2, 3)과 (3, 2) →
- (2, 3)과 (3,) →
- (3,)과 (3, 1) →

대응하는 열/행을 제외한 나머지 index는
앞에서 뒤쪽으로 axis=0, 1, ... 로 사용된다

- 배열과 스칼라 값의 연산은 broadcast 연산이 된다

- (3, 2)와 2 → (3, 2)

dot 연산의 예



▶ [예제27] dot은 matmul과 다음과 같은 차이가 있다

A	1	2	3
B	4	5	6
C	1	1	1
D	2	2	2

shape : [2, 2, 3]

공간, 면 위치

1	2
3	4
5	6
3	4
1	1
1	2
2	2

shape : [2, 3, 2]

행, 열 위치

A*1	A*2	B*1	B*2
A*3	A*4	B*3	B*4

C*1	C*2	D*1	D*2
C*3	C*4	D*3	D*4

A	0,0
B	0,1
C	1,0
D	1,1

1	0,0
2	0,1
3	1,0
4	1,1

- A*1 → 0, 0, 0, 0
- B*2 → 0, 1, 0, 1

배열 파일 입출력



▶ numpy의 배열을 파일로 저장하고 불러오는 함수는 다음과 같다

함수	기능
np.save()	<ul style="list-style-type: none">▪ 1개 배열을 npy 파일로 저장 (NumPy format, binary file)
np.savez()	<ul style="list-style-type: none">▪ 여러 개 배열을 압축 안 된 npz 파일로 저장▪ 각 배열에 이름을 부여하여 저장
np.save_compressed()	<ul style="list-style-type: none">▪ 압축 된 npz 파일로 저장, 각 배열에 이름을 부여하여 저장
np.load()	<ul style="list-style-type: none">▪ save, savez, save_compressed로 저장된 npy, npz 파일 불러오기▪ npz를 읽어온 경우, 이름을 이용하여 indexing▪ npz type : numpy.lib.npyio.NpzFile, close() 할것
np.savetxt()	<ul style="list-style-type: none">▪ 1D/2D array_like를 텍스트 파일로 저장▪ delimiter, newline, encoding, fmt 등 지정 가능 (fmt='%.1.2f')
np.loadtxt()	<ul style="list-style-type: none">▪ Text 파일 불러오기, delimiter, dtype 등 지정 가능

- 각 함수의 자세한 사용 법은 필요시 Numpy API를 참조하며, 예제를 사용하여 알아보자

배열 파일 입출력 예제



▶ [예제29] 다음 배열 파일 입출력 예시를 실행하고 파일을 확인해 보자

```
np.save(path+"a1.npy", a1)
r = np.load(path+"a1.npy")
printary(r)
```

```
np.savez(path+"an.npz", A1=a1, A2=a2, A3=a3)
r = np.load(path+"an.npz")
printary(r['A1'], r['A2'], r['A3'])
```

```
np.savez_compressed(path+"c_an.npz", A1=a1, A2=a2, A3=a3)
r = np.load(path+"c_an.npz")
printary(r['A1'], r['A2'], r['A3'])
```

```
np.savetxt(path+"a1.txt", a1, fmt='%d')
r = np.loadtxt(path+"a1.txt", dtype=np.int32)
printary(r)
```

	a1.npy	79KB
	a1.txt	126KB
	an.npz	107KB
	c_an.npz	40KB

np.nan, np.inf



▶ [예제15] 다음 numpy의 version, nan, inf에 대한 속성과 함수를 살펴보자

속성, 함수	설명
np.__version__	▪ numpy의 버전
np.nan	▪ not a number, ‘nan’으로 출력됨
np.inf	▪ infinite, ‘inf’으로 출력됨
np.isnan()	▪ np.nan 값을 갖는 항을 True, 아닌 항을 False로 하여 반환
np.isinf()	▪ np.inf 값을 갖는 항을 True, 아닌 항을 False로 하여 반환

```
print(np.__version__)
1.17.2
print(np.nan, type(np.nan))
nan <class 'float'>
print(np.inf, type(np.inf))
inf <class 'float'>

a = np.array([np.nan, 1, 0, -1, np.inf], dtype=np.float16)
print(np.isnan(np.nan), np.isnan(a))
True [ True False False False False]
print(np.isinf(np.inf), np.isinf(a))
True [False False False False  True]
```

- nan, inf가 포함된 경우 연산 시 원치 않는 상황이 발생할 수 있으므로 주의 해야함