

Python & Data Engineering

인공지능 직무전환자 과정

- Day 3
Numpy & Pandas



Sanghyun Seo

Course Descriptions

- Python & Data Engineering

Day 1 – Python Basic	Day 2 – Advanced Python	Day 3 – Numpy, Pandas	Day 4 – Data Analysis, Visualization
<ul style="list-style-type: none">• Getting started• Introduction to Python• Data Type & Variable• Flow control• Function• Python programming practice	<ul style="list-style-type: none">• Review• File• Class• Exception Handling• Advanced python	<ul style="list-style-type: none">• Review• Module, Package• Numpy Basic• Advanced Numpy• Pandas Basic• Advanced Pandas	<ul style="list-style-type: none">• Review• Introduction to machine learning• Data preprocessing• Visualization• Course summarization

Contents

1. Module & Package

2. Numpy Basic

- ndarray
- Shape manipulation
- Indexing & slicing

3. Advanced Numpy

- 주요 메소드

4. Pandas Basic

- Series & DataFrame
- Indexing & slicing

5. Advanced Pandas

- 주요 메소드

1. Module & Package

모듈 (module)

- 모듈(module)
 - 함수나 변수, 클래스 등을 가진 파일 (.py)
 - 모듈 안에는 함수, 클래스 또는 변수들이 정의되어 있음
 - 파이썬은 많은 표준 라이브러리 모듈을 제공

```
import math
```

```
math.factorial(4)
```

24

```
from math import factorial
```

```
factorial(4)
```

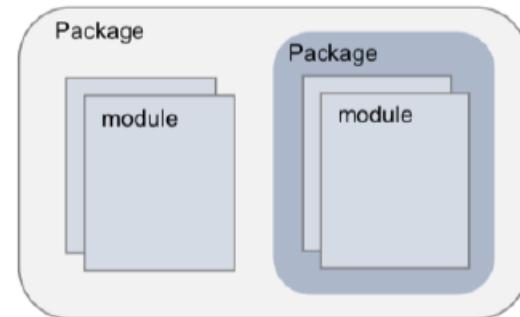
24

모듈 (module)

- 패키지(package)

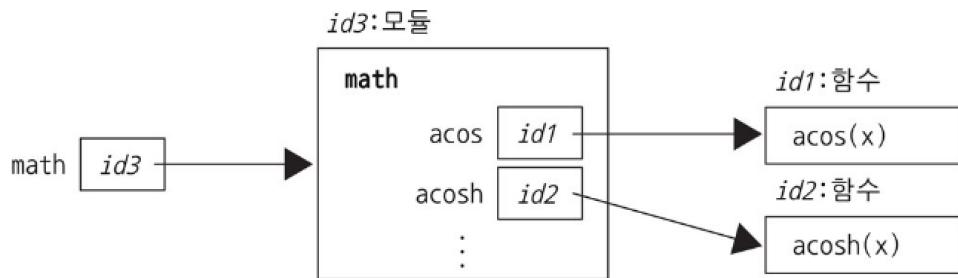
- 모듈을 효율적으로 관리하기 위한 모듈의 상위 개념
- 공동 작업이나 코드의 유지 보수 등에 유리

```
import 패키지.모듈  
import 패키지.모듈.변수  
import 패키지.모듈.함수  
import 패키지.모듈.클래스  
  
from 패키지.모듈 import 변수/함수/클래스
```



모듈 (module)

- 모듈(module)
 - import math
→ 모듈 객체를 참조하는 math라는 변수 생성
 - help(): 모듈의 정보 제공

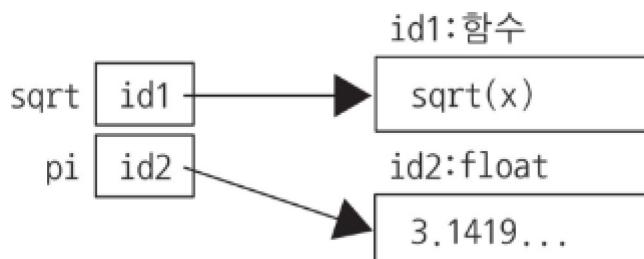


```
1 import math
1 type(math)
module
1 help(math)
Help on built-in module math:
NAME
    math
DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.
FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.
```

모듈 (module)

- 모듈(module)
 - from math import sqrt, pi
→sqrt 및 pi 변수만 생성



```
1 del math
2 from math import sqrt, pi

1 math.sqrt(9)

-----
NameError: name 'math' is not defined                                     Traceback (most recent call last)
<ipython-input-6-c6664062629c> in <module>()
      1 math.sqrt(9)

NameError: name 'math' is not defined

SEARCH STACK OVERFLOW

1 sqrt(9)

3.0

1 pi

3.141592653589793
```

모듈 (module)

- 모듈(module)
 - from math import *
 - math 모듈의 모든 구성요소 로드

```
1 del sqrt
2 sqrt(8)

-----
NameError                                                 Traceback (most recent call last)
<ipython-input-9-30a07089954b> in <module>()
      1 del sqrt
----> 2 sqrt(8)

NameError: name 'sqrt' is not defined

SEARCH STACK OVERFLOW

1 from math import *
2 sqrt(8)

2.8284271247461903

1 pi

3.141592653589793
```

Google Python Style Guide

- Imports

- Use import statements for packages and modules only, not for individual classes or functions
- Note that there is an explicit exemption for imports from the typing module

2.2.4 Decision

- Use `import x` for importing packages and modules.
- Use `from x import y` where `x` is the package prefix and `y` is the module name with no prefix.
- Use `from x import y as z` if two modules named `y` are to be imported or if `y` is an inconveniently long name.
- Use `import y as z` only when `z` is a standard abbreviation (e.g., `np` for `numpy`).

For example the module `sound.effects.echo` may be imported as follows:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

Google Python Style Guide

- Packages

- Import each module using the full pathname location of the module

2.3.3 Decision

All new code should import each module by its full package name.

Imports should be as follows:

Yes:

```
# Reference absl.flags in code with the complete name (verbose).
import absl.flags
from doctor.who import jodie

FLAGS = absl.flags.FLAGS
```

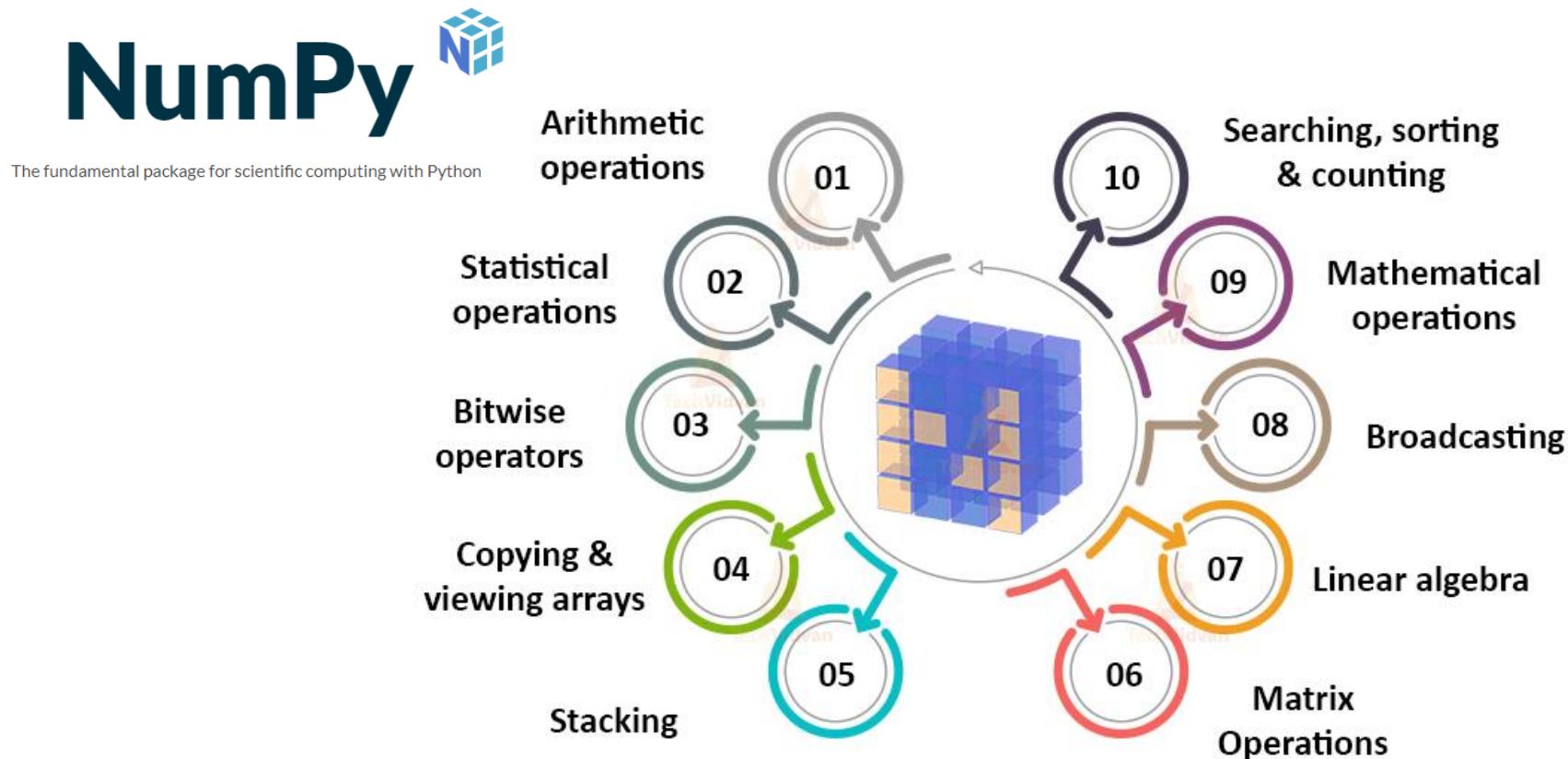
```
# Reference flags in code with just the module name (common).
from absl import flags
from doctor.who import jodie

FLAGS = flags.FLAGS
```

2. Numpy Basic

넘파이 (Numpy)

- 넘파이 (Numpy)
 - Numerical Python의 약자로서 산술계산에 특화된 라이브러리



넘파이 (Numpy)

- numpy의 특징
 - ndarray(다차원 배열 객체)
 - numpy에서 제공하는 대규모의 데이터 집합을 담을 수 있는 자료 구조로서 N차원의 배열 객체를 의미
 - 빠르고 효율적인 메모리 사용, 유연한 브로드캐스팅 지원
 - 디스크로부터 배열 기반의 데이터를 읽거나 쓰기 용이
 - C, C++, 포트란 등으로 쓰여진 코드를 통합 가능
 - 선형대수 계산, 푸리에 변환, 난수 생성기 등 다양한 기능 포함

ndarray

- list vs. ndarray
 - python list와 numpy ndarray의 연산속도 비교

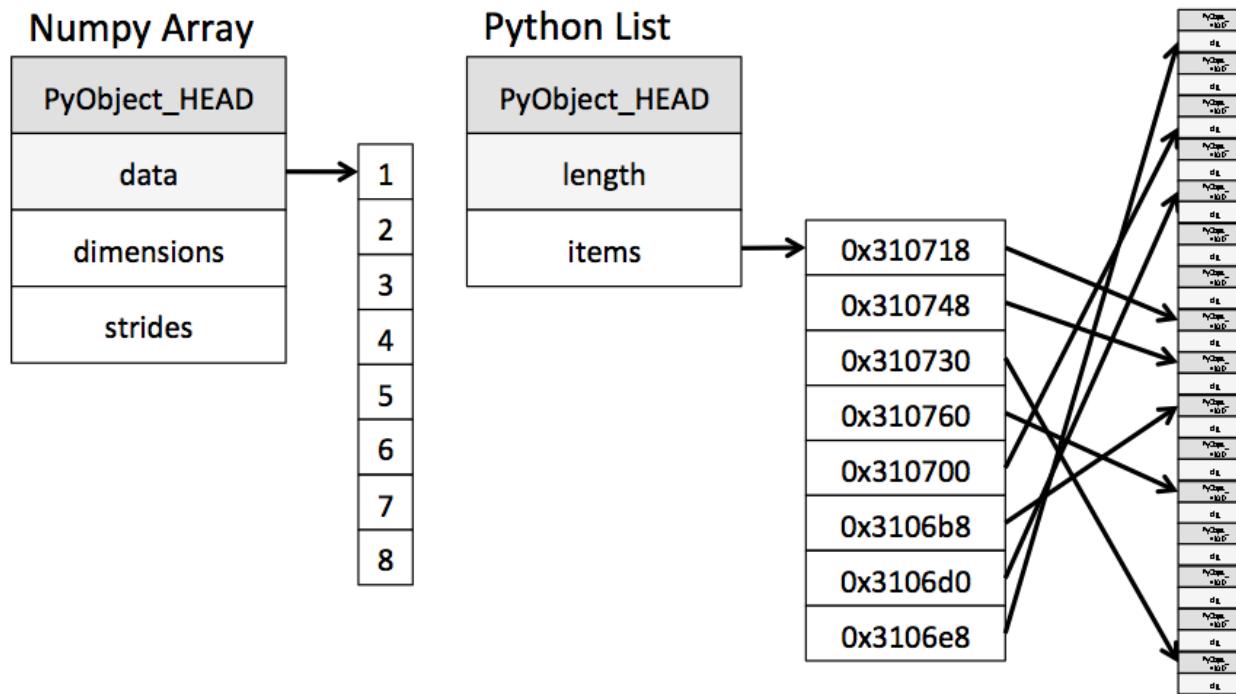
```
n = 100000000
numpy_arr = np.arange(n)
python_list = list(range(n))

%%time
python_list = [x**3+10 for x in python_list]
Wall time: 5.17 s

%%time
numpy_arr = numpy_arr**3+10
Wall time: 62 ms
```

ndarray

- list vs. ndarray
 - python list와 numpy ndarray의 연산속도 비교
 - numpy는 연속된 메모리 블록에 데이터 저장
 - 같은 종류의 데이터를 담음



ndarray

- **dtype**

- 배열에 담긴 원소의 자료형 (ndarray는 같은 자료형으로 구성됨)
- dtype으로 데이터 타입을 명시하지 않은 경우 자료형을 추론

```
arr = np.array([10, 20, 30, 40])  
arr
```

```
array([10, 20, 30, 40])
```

```
arr.dtype
```

```
dtype('int32')
```

```
arr2 = np.array([[0.1, 0.6], [-2, 6]])  
arr2
```

```
array([[ 0.1,  0.6],  
       [-2. ,  6. ]])
```

```
arr2.dtype
```

```
dtype('float64')
```

ndarray

- **dtype**
 - astype 메소드를 이용하여 dtype을 명시적으로 변환 가능

```
float_arr = arr.astype(np.float64)  
float_arr
```

```
array([10., 20., 30., 40.])
```

```
float_arr.dtype
```

```
dtype('float64')
```

ndarray

- **size**
 - 배열에 있는 원소의 전체 개수
- **ndim**
 - 배열의 차원의 개수
- **shape**
 - 배열의 각 차원의 크기, 튜플 형태로 반환

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
arr
array([1, 2, 3, 4, 5, 6, 7, 8])

arr.size
8

arr.ndim
1

arr.shape
(8,)
```

ndarray

- **size**
 - 배열에 있는 원소의 전체 개수
- **ndim**
 - 배열의 차원의 개수
- **shape**
 - 배열의 각 차원의 크기, 튜플 형태로 반환

```
arr2 = np.array([[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]])  
arr2  
  
array([[ 1,  2,  3,  4,  5,  6],  
       [ 7,  8,  9, 10, 11, 12]])  
  
arr2.size  
  
12  
  
arr2.ndim  
  
2  
  
arr2.shape  
  
(2, 6)
```

연습문제 1

- Q1 ndarray의 dtype, size, ndim, shape을 예상해보기

```
arr3 = np.array([[[10.          , 10.52631579, 11.05263158, 11.57894737, 12.10526316],
                  [12.63157895, 13.15789474, 13.68421053, 14.21052632, 14.73684211]],
                  [[15.26315789, 15.78947368, 16.31578947, 16.84210526, 17.36842105],
                   [17.89473684, 18.42105263, 18.94736842, 19.47368421, 20.        ]]])
arr3

array([[[10.          , 10.52631579, 11.05263158, 11.57894737,
         12.10526316],
        [12.63157895, 13.15789474, 13.68421053, 14.21052632,
         14.73684211]],

       [[15.26315789, 15.78947368, 16.31578947, 16.84210526,
         17.36842105],
        [17.89473684, 18.42105263, 18.94736842, 19.47368421,
         20.        ]]])
```

ndarray 생성

- array 함수 → np.array(데이터, dtype=[data type])
 - 기존에 있던 데이터(자료형)을 이용하여 새로운 배열 생성
 - 리스트를 이용한 배열 생성

```
arr = np.array([10, 20, 30])
arr
```

```
array([10, 20, 30])
```

```
arr2 = np.array([10, 20, 30], dtype=np.float16)
arr2
```

```
array([10., 20., 30.], dtype=float16)
```

ndarray 생성

- array 함수 → np.array(데이터, dtype=[data type])
 - 기존에 있던 데이터(자료형)을 이용하여 새로운 배열 생성
 - 튜플, range 함수를 이용한 배열 생성

```
arr3 = np.array(((1, 0), (0, 1)), dtype=np.float32)  
arr3
```

```
array([[1., 0.],  
       [0., 1.]], dtype=float32)
```

```
arr4 = np.array(range(20))  
arr4
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  
   12, 13, 14, 15, 16,  
   17, 18, 19])
```

ndarray 생성

- array 함수 → np.array(데이터, dtype=[data type])
 - 기존에 있던 데이터(자료형)을 이용하여 새로운 배열 생성
 - 리스트+튜플, 요소 길이가 서로 다른 튜플을 이용한 배열생성

```
arr5 = np.array([(10, 20), (40, 50)])
arr5
```

```
array([[10, 20],
       [40, 50]])
```

```
arr6 = np.array(((1, 2), (3)))
arr6
```

```
array([(1, 2), 3], dtype=object)
```

```
arr6.size
```

```
2
```

ndarray 생성

- 배열 생성 함수
 - 넘파이의 표준 배열 함수 참고
 - zeros: 모두 0으로 초기화
 - ones: 모두 1로 초기화
 - full: 특정 값으로 모두 채워 초기화
 - empty: 초기화 되지 않은 배열을 생성
 - identity, eye: NxN 크기의 단위행렬
 - _likes: 주어진 어떤 배열과 같은 shape의 배열 생성
 - range: range 함수와 유사, 범위와 stepsize 설정
 - linspace: range 함수와 유사, sample의 개수 설정
 - 자료형을 명시하지 않으면 dtype은 float64로 설정됨

ndarray 생성

- 배열 생성 함수

- zeros

```
np.zeros((5))
```

```
array([0., 0., 0., 0., 0.])
```

```
np.zeros((2, 4), dtype=np.int8)
```

```
array([[0, 0, 0, 0],  
       [0, 0, 0, 0]], dtype=int8)
```

- ones

```
np.ones((3,3))
```

```
array([[1., 1., 1.],  
       [1., 1., 1.],  
       [1., 1., 1.]])
```

ndarray 생성

- 배열 생성 함수
 - full / empty

```
np.full((4), 5)
```

```
array([5, 5, 5, 5])
```

```
np.full((2, 5), -1.0)
```

```
array([[-1., -1., -1., -1., -1.],
       [-1., -1., -1., -1., -1.]])
```

```
np.full((2, 3, 4), np.inf)
```

```
array([[[inf, inf, inf, inf],
        [inf, inf, inf, inf],
        [inf, inf, inf, inf]],
```

```
      [[inf, inf, inf, inf],
       [inf, inf, inf, inf],
       [inf, inf, inf, inf]]])
```

```
np.empty((2,3), dtype=np.float64)
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

ndarray 생성

- 배열 생성 함수
 - identity
 - NxN 정방 행렬만 생성 가능
 - eye
 - NxM 행렬 생성 가능
 - k: 1값의 시작 위치

```
np.identity(5, dtype=int)  
  
array([[1, 0, 0, 0, 0],  
       [0, 1, 0, 0, 0],  
       [0, 0, 1, 0, 0],  
       [0, 0, 0, 1, 0],  
       [0, 0, 0, 0, 1]])
```

```
np.eye(5, dtype=int)
```

```
array([[1, 0, 0, 0, 0],  
       [0, 1, 0, 0, 0],  
       [0, 0, 1, 0, 0],  
       [0, 0, 0, 1, 0],  
       [0, 0, 0, 0, 1]])
```

```
np.eye(5, 10, dtype=int, k=5)
```

```
array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

ndarray 생성

- 배열 생성 함수
 - _likes

```
zeros_like
```

```
ones_like
```

```
full_like
```

```
arr1 = np.array([[1, 2, 3, 1], [2, 4, 5, 6]])  
arr1
```

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

```
arr2 = np.ones_like(arr1)  
arr2
```

```
array([[1, 1, 1, 1],  
       [1, 1, 1, 1]])
```

ndarray 생성

- 배열 생성 함수
 - `arrange`
 - 파이썬 내장함수 `range`와 유사, ndarray를 반환
 - 세번째 인자는 step size(간격)

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
np.arange(-3, 3)
```

```
array([-3, -2, -1, 0, 1, 2])
```

```
np.arange(3,50,5)
```

```
array([ 3,  8, 13, 18, 23, 28, 33, 38, 43, 48])
```

```
np.arange(0, 1, 0.2)
```

```
array([0. , 0.2, 0.4, 0.6, 0.8])
```

ndarray 생성

- 배열 생성 함수
 - linspace
 - 범위 내에서 주어진 sample의 개수만큼 생성

```
np.linspace(0, 1, 6)
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

ndarray 생성

- 난수생성
 - numpy.random 모듈을 이용하여 다양한 종류의 확률분포로부터 표본값을 생성
 - np.random.seed()
 - 난수 생성기의 시드를 지정
 - np.random.rand()
 - [0, 1) 범위의 균등분포에서 표본을 추출
 - np.random.randn()
 - 표준편차 1, 평균 0인 정규분포에서의 표본 추출
 - np.random.randint()
 - 주어진 범위 내에서 임의의 난수 추출
 - np.random.permutation()
 - 순서를 임의로 바꾸거나 임의의 순열을 반환
 - np.random.shuffle()
 - 리스트나 배열의 순서를 뒤섞음

ndarray 생성

- 난수생성
 - np.random.seed()
 - 난수 발생을 위한 seed를 고정하여 reproduction시 활용
 - visualization을 위한 matplotlib 활용

```
import matplotlib.pyplot as plt
```

np.random.seed - seed 고정

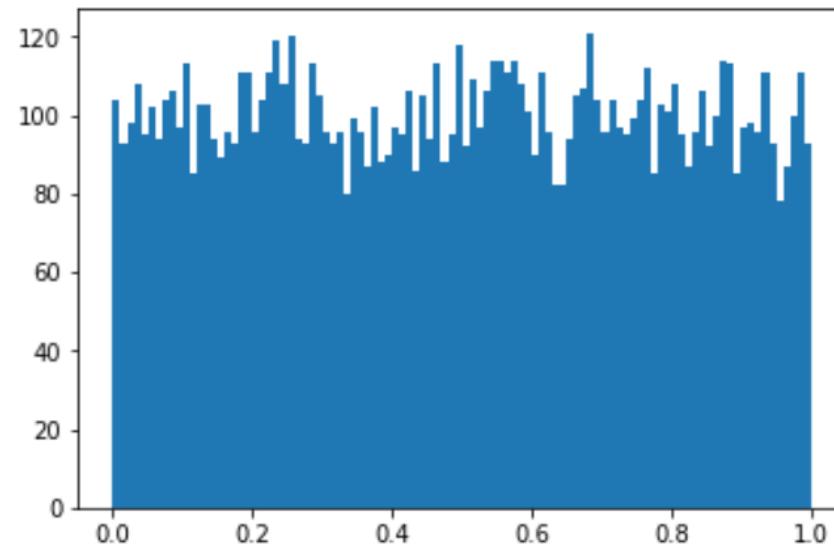
```
np.random.seed(1)  
np.random.rand(2)
```

```
array([0.417022 , 0.72032449])
```

ndarray 생성

- 난수생성
 - np.random.rand()

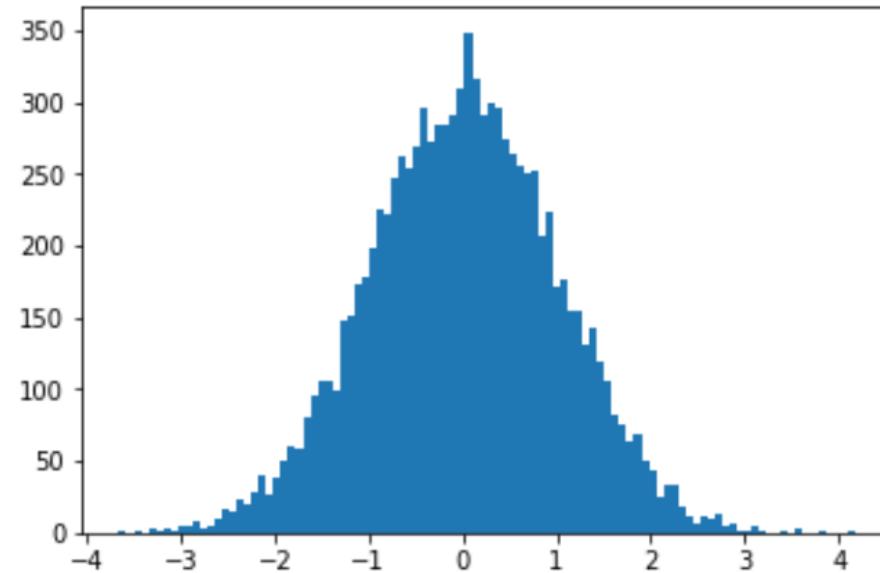
```
data = np.random.rand(10000)
plt.hist(data, bins=100)
plt.show()
```



ndarray 생성

- 난수생성
 - np.random.randn()

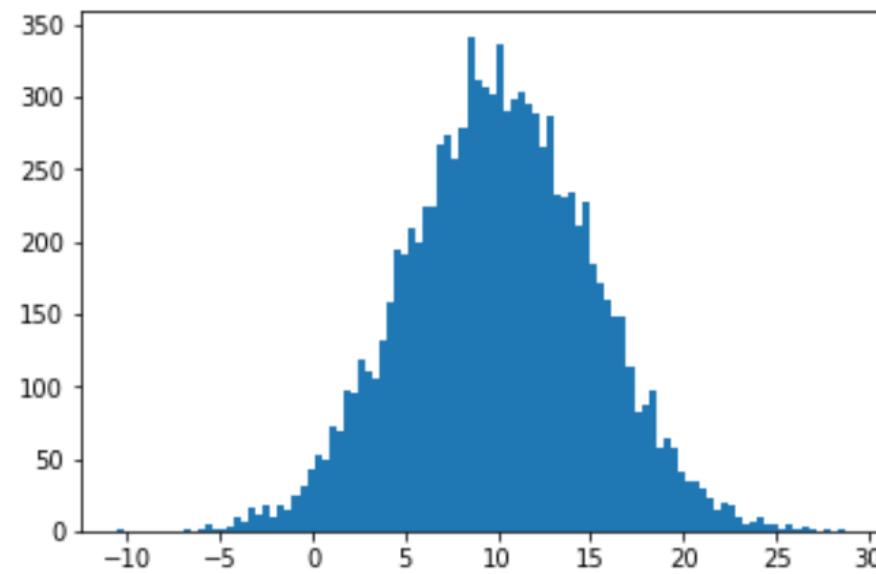
```
data = np.random.randn(10000)
plt.hist(data, bins=100)
plt.show()
```



ndarray 생성

- 난수생성
 - np.random.randn()
 - 평균과 분산

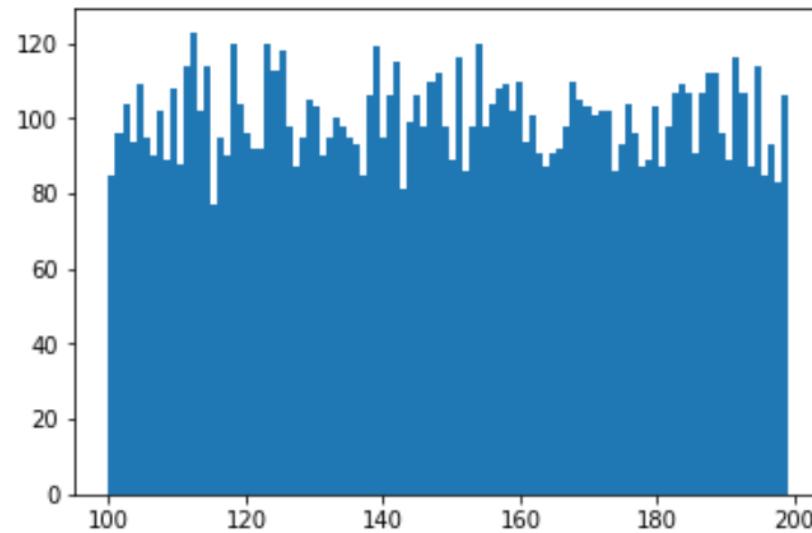
```
data = 5*np.random.randn(10000)+10    # numpy.random.no
plt.hist(data, bins=100)
plt.show()
```



ndarray 생성

- 난수생성
 - np.random.randint()

```
data = np.random.randint(100, 200, 10000)
plt.hist(data, bins=100)
plt.show()
```



shape manipulation

- flatten
 - n차원의 ndarray를 1차원으로 변형

1	2	3
4	5	6
7	8	9



1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

```
arr = np.zeros((3,2))
arr
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

```
arr.flatten()
```

```
array([0., 0., 0., 0., 0., 0.])
```

shape manipulation

- reshape
 - `np.reshape(arr, shape)`
 - `arr.reshape(shape)`
 - 이미 존재하는 ndarray를 원하는 shape으로 변형하는 함수

```
arr = np.arange(12)
arr

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

arr.reshape(3, 4)

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

shape manipulation

- **reshape**

- order: {'C', 'F', 'A'}

- 'C'는 C언어의 인덱스의 규칙으로, 'F'는 Fortran의 인덱스 규칙으로 읽고 쓰기 옵션 (일반적으로는 C형태 사용)

```
arr.reshape(2, 6)
```

```
array([[ 0,  1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10, 11]])
```

```
arr.reshape(2, 6, order='F')
```

```
array([[ 0,  2,  4,  6,  8, 10],  
       [ 1,  3,  5,  7,  9, 11]])
```

shape manipulation

- **reshape**

- -1을 사용하면 shape을 명시하지 않아도 자동으로 채워줌 (단, 1개의 차원이 남아있는 경우만 가능)

```
arr = np.arange(20)
arr

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
       12, 13, 14, 15, 16,
       17, 18, 19])

arr.reshape(-1, 10)

array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])

arr.reshape(5, -1)

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

shape manipulation

- transpose
 - 다차원 배열의 전치를 수행

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{|c|c|c|} \hline 1 & 4 & 7 \\ \hline 2 & 5 & 8 \\ \hline 3 & 6 & 9 \\ \hline \end{array}$$

A

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

shape manipulation

- transpose
 - np.transpose(arr, axis)
 - arr.transpose(axis)
 - 다차원 배열의 전치를 수행

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{|c|c|c|} \hline 1 & 4 & 7 \\ \hline 2 & 5 & 8 \\ \hline 3 & 6 & 9 \\ \hline \end{array}$$

A^T

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

shape manipulation

- transpose

```
arr = np.arange(20).reshape(4, 5)
arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
print(arr.transpose().shape)
arr.transpose()
```

```
(5, 4)
```

```
array([[ 0,  5, 10, 15],
       [ 1,  6, 11, 16],
       [ 2,  7, 12, 17],
       [ 3,  8, 13, 18],
       [ 4,  9, 14, 19]])
```

shape manipulation

- transpose

- axis를 지정하지 않으면
→ arr.shape = (i[0], i[1], ..., i[n-1])
→ arr.transpose().shape = (i[n-1], i[n-2], ..., i[0])

```
arr = np.arange(30).reshape(3,2,5)
arr.shape
```

```
(3, 2, 5)
```

```
print(arr.transpose().shape)
```

```
(5, 2, 3)
```

shape manipulation

- T

- transpose와 같은 역 할을 수행하는 ndarray의 attribute
- 단, T의 경우 axis를 지정할 수 없음

```
x = np.arange(24).reshape((-1,3,4))
print(x.shape)
x

(2, 3, 4)
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],

      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]]])

print(x.T.shape)
x.T

(4, 3, 2)
array([[[ 0, 12],
       [ 4, 16],
       [ 8, 20]],

      [[ 1, 13],
       [ 5, 17],
       [ 9, 21]],

      [[ 2, 14],
       [ 6, 18],
       [10, 22]],

      [[ 3, 15],
       [ 7, 19],
       [11, 23]]])
```

Practice 2

- Q1 numpy를 이용하여 10~20사이 짹수배열 생성
- Q2 주어진 배열을 이용하여 제시된 출력 결과들로 변형

```
arr = np.arange(30).reshape(2,3,5)
arr
```

```
array([[[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]],

      [[15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]]])
```

indexing & slicing

- 1차원 indexing & slicing
 - 파이썬의 리스트와 유사

```
arr1d = np.arange(8)  
arr1d
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
arr1d[1]
```

```
1
```

```
arr1d[-1]
```

```
7
```

```
arr1d[:4]
```

```
array([0, 1, 2, 3])
```

indexing & slicing

- 브로드캐스팅 (broadcasting)
 - 다른 모양의 배열 간 산술연산을 보다 쉽게 수행할 수 있도록 해주는 numpy의 기능
 - **python list vs. numpy ndarray**

```
lst = list(range(6))
lst
```

```
[0, 1, 2, 3, 4, 5]
```

```
lst[2:5] = -1 # 브로드캐스팅 x
lst
```

```
-----
TypeError                      Traceback (most recent call last)
<ipython-input-92-aaed574afcab> in <module>()
----> 1 lst[2:5] = -1 # 브로드캐스팅 x
      2 lst
```

```
TypeError: can only assign an iterable
```

```
lst[3] = -1
lst
```

indexing & slicing

- 브로드캐스팅 (broadcasting)
 - python list vs. numpy ndarray

```
arr1d = np.arange(8)
arr1d
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
arr1d[3:6] = 100 # 브로드캐스팅
```

```
arr1d
```

```
array([ 0, 1, 2, 100, 100, 100, 6, 7])
```

indexing & slicing

- 뷰 (view)

- numpy의 슬라이싱은 원본데이터의 뷰를 제공 → 데이터를 새롭게 복사해 오는 것이 아니라 기존의 데이터와 연결 (얕은 복사)

```
arr_part = arr1d[:3]      → 슬라이싱  
arr_part
```

```
array([0, 1, 2])
```

```
arr_part[1:] = -1        → 새로운 값으로 변경  
arr_part
```

```
array([ 0, -1, -1])
```

```
arr1d                  → 원본 데이터가 변경되어 있음
```

```
array([ 0, -1, -1, 100, 100, 100, 6, 7])
```

indexing & slicing

- 뷰 (view)
 - 리스트의 슬라이싱은 데이터를 깊은 복사

```
lst_part = lst[2:]      → 슬라이싱
```

```
[2, -1, 4, 5]
```

```
lst_part[3] = 100      → 새로운 값으로 변경
```

```
[2, -1, 4, 100]
```

```
lst                  → 원본 데이터에 변경 없음
```

```
[0, 1, 2, -1, 4, 5]
```

indexing & slicing

- 뷰 (view)

- numpy의 원본 데이터를 훼손하지 않으면서 indexing 또는 slicing을 활용하기 위해서는 깊은 복사를 수행 후 이후 절차 진행
→ `arr.copy()`

```
new_arr = arr1d.copy()
new_arr

array([ 0, -1, -1, 100, 100, 100,    6,    7])
```

```
new_arr_part = new_arr[1:5]
new_arr_part[0:2] = 0
new_arr_part
```

```
array([ 0,  0, 100, 100])
```

```
new_arr

array([ 0,  0,  0, 100, 100, 100,    6,    7])
```

```
arr1d

array([ 0, -1, -1, 100, 100, 100,    6,    7])
```

indexing & slicing

- 다차원 배열의 인덱싱

```
arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

```
arr2d[0]
```

```
array([0, 1, 2, 3, 4])
```

```
arr2d[1][2] → 재귀적으로 접근
```

```
7
```

```
arr2d[1, 2] → 콤마(,)를 이용한 인덱싱
```

```
7
```

indexing & slicing

- 다차원 배열의 슬라이싱

```
arr2d = np.arange(20).reshape(4, -1)  
arr2d
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

```
arr2d[:3][:2] → 재귀적으로 접근
```

```
array([[0, 1, 2, 3, 4],  
       [5, 6, 7, 8, 9]])
```

```
arr2d[:3, :2] → 콤마(,)를 이용한 슬라이싱
```

```
array([[ 0,  1],  
       [ 5,  6],  
       [10, 11]])
```

indexing & slicing

- 다차원 배열의 슬라이싱

```
arr3d = np.arange(30).reshape(2, 3, -1)
arr3d
```

```
array([[[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]],

      [[15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]]])
```

```
arr3d[1, 0, 2]
```

```
17
```

```
arr3d[:, :, :3]
```

```
array([[[ 0,  1,  2],
       [ 5,  6,  7],
       [10, 11, 12]]])
```

indexing & slicing

- boolean indexing

- 불리언 배열: 불리언 값으로 이루어진 배열
- 불리언 인덱싱: True에 해당하는 위치에 있는 값만 반환

```
arr = np.array([True, False])
arr.dtype
```

```
dtype('bool')
```

```
arr = np.array([0, 1, 2, 3, 4], int)
arr[[True, False, True, False, True]]
```

```
array([0, 2, 4])
```

indexing & slicing

- boolean indexing

- 불리언 인덱싱 후 해당 값을 다시 인덱싱/슬라이싱으로 활용 가능

```
arr = np.array([10, 20, 30, 40, 50, 60], int)  
arr
```

```
array([10, 20, 30, 40, 50, 60])
```

```
arr % 3 == 0
```

```
array([False, False, True, False, False, True])
```

```
arr[arr%3==0]
```

```
array([30, 60])
```

indexing & slicing

- fancy indexing
 - 정수가 담긴 ndarray나 리스트로 특정 위치에 있는 값을 가져올 수 있음
 - 인덱스 값이 주어진 순서대로 가져오게 됨

```
arr = np.arange(10, 20)
arr

array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

arr[[0, 2, 4, 6]]

array([10, 12, 14, 16])

arr[[3, 0, 1]]

array([13, 10, 11])
```

indexing & slicing

- fancy indexing

- 정수가 담긴 ndarray나 리스트로 특정 위치에 있는 값을 가져올 수 있음
- 인덱스 값이 주어진 순서대로 가져오게 됨

```
arr2d = np.arange(20).reshape(4, 5)
arr2d
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
arr2d[[0, 2]]
```

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
```

```
arr2d[[0, 1], [4]]
```

```
array([4, 9])
```

indexing & slicing

- fancy indexing
 - 순서에 따라 일종의 좌표처럼 인덱싱 수행

```
arr2d[[0, 1], [4, 3]] → (0, 4), (1, 3)에 대응되는 요소 반환
```

```
array([4, 8])
```

```
arr2d[[0, 1, 2], [4, 3, 1]]
```

```
array([ 4, 8, 11])
```

```
arr2d[[0, 1, 2]] [:, [4, 3, 1]]
```

```
array([[ 4, 3, 1],
       [ 9, 8, 6],
       [14, 13, 11]])
```

Practice 3

- Q1 numpy를 이용하여 주어진 배열 변형
- Q2 불리언 인덱싱 활용

```
arr = np.arange(24).reshape(2,4,3)
arr

array([[[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]],

       [[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]]])
```

```
arr = np.arange(30).reshape(3,2,5)
arr

array([[[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]],
       [[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]],
       [[20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])]

cond = np.array(['a', 'b', 'c'])
```

3. Advanced Numpy

산술 연산

- 벡터화 (vectorization)
 - 배열은 for 문을 작성하지 않고 데이터를 일괄 처리 가능
 - 같은 크기의 배열 간의 산술 연산은 배열의 각 요소 단위로 적용

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

arr + arr
array([[ 2,  4,  6],
       [ 8, 10, 12]])

arr - arr
array([[0, 0, 0],
       [0, 0, 0]])

arr * arr
array([[ 1,  4,  9],
       [16, 25, 36]])

arr / arr
array([[1., 1., 1.],
       [1., 1., 1.]])
```

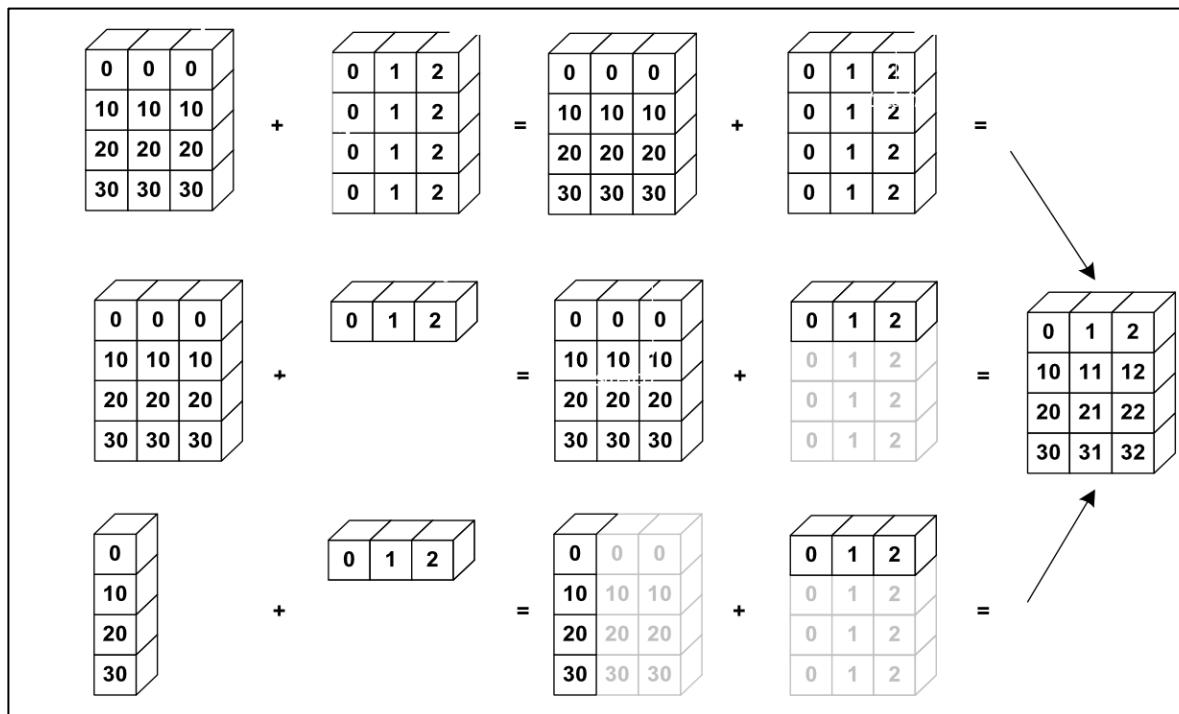
arr**arr

```
array([[    1,     4,    27],
       [ 256, 3125, 46656]], dtype=int32)
```

산술 연산

- 브로드캐스팅

- 다른 shape의 배열 간의 산술 연산 수행
- 브로드캐스팅이 가능하려면 연산을 수행하는 축을 제외한 나머지 축의 shape이 일치하거나 둘 중 하나의 길이가 1이여야 함



산술 연산

- 브로드캐스팅
 - 스칼라 인자: 모든 요소에 각각 적용

```
arr + 10
```

```
array([[11, 12, 13],  
       [14, 15, 16]])
```

```
10 - arr
```

```
array([[9, 8, 7],  
       [6, 5, 4]])
```

```
arr*3
```

```
array([[ 3,  6,  9],  
       [12, 15, 18]])
```

산술 연산

- 브로드캐스팅
 - 배열 : shape에 맞게 적용
 - $(2,3) + (1,3) \rightarrow (2,3)$

```
arr2 = np.array([100, 200, 300])
arr2
array([100, 200, 300])

arr
array([[1, 2, 3],
       [4, 5, 6]])

arr2
array([100, 200, 300])

arr + arr2
array([[101, 202, 303],
       [104, 205, 306]])
```

산술 연산

- 브로드캐스팅
 - 배열 : shape에 맞게 적용
 - $(2,3) + (2,1) \rightarrow (2,3)$

```
arr3 = np.array([[100],[200]])  
arr3
```

```
array([[100],  
       [200]])
```

```
arr
```

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

```
arr3
```

```
array([[100],  
       [200]])
```

```
arr + arr3
```

```
array([[101, 102, 103],  
       [204, 205, 206]])
```

유니버셜 함수 (ufunc)

- 유니버셜 함수
 - ndarray 안에 있는 데이터 요소별 연산을 수행하는 함수
 - 하나 이상의 스칼라 값을 받아서 하나 이상의 스칼라 값을 반환하는 간단한 함수를 고속으로 수행할 수 있는 벡터화 된 함수

유니버셜 함수 (ufunc)

- 단항 유니버셜 함수

Function	설명
abs, fabs	각 원소의 절대값. 복소수가 아닌 경우 fabs를 쓰면 빠른 연산이 가능
sqrt	각 원소의 제곱근 계산($arr^{**0.5}$)
square	각 원소의 제곱을 계산(arr^{**2})
exp	각 원소에 지수 e^x 계산
log, log10, log2, log1p	자연로그, 밀10인 로그, 밀2인 로그, $\log(1+x)$
sign	각 원소의 부호(양수 1, 영 0, 음수 -1)
ceil	각 원소의 값보다 같거나 큰 정수 중 가장 작은 값
floor	각 원소의 값보다 같거나 작은 정수 중 가장 큰 값
rint	각 원소의 소수자리를 반올림
modf	각 원소의 몫과 나머지를 각각의 배열로 반환
isnan	각 원소가 NaN인지 아닌지. 불리언 배열로 반환
isfinite, isinf	각 원소가 유한한지, 무한한지. 불리언 배열로 반환
cos, cosh, sin, sinh, tan, tanh	일반 삼각함수, 쌍곡삼각함수
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	역삼각함수

유니버셜 함수 (ufunc)

- 단항 유니버셜 함수

```
arr = np.arange(-3, 3).reshape(3, -1)
arr
```

```
array([[-3, -2],
       [-1,  0],
       [ 1,  2]])
```

```
np.exp(arr)
```

```
array([[0.04978707, 0.13533528],
       [0.36787944, 1.          ],
       [2.71828183, 7.3890561 ]])
```

```
np.floor(arr)
```

```
array([[-3., -2.],
       [-1.,  0.],
       [ 1.,  2.]])
```

유니버설 함수 (ufunc)

- 이항 유니버설 함수

Function	설명
add	두 배열의 같은 위치의 원소끼리 합함
subtract	첫번째 배열의 원소에서 두번째 배열의 원소를 뺌
multiply	같은 위치의 원소끼리 곱함
divide, floor_divide	첫번째 배열의 원소를 두번째 배열의 원소로 나눔. floor는 몫만 취함
power	첫번째 배열의 원소를 두번째 배열의 원소만큼 제곱함
maximum, fmax	각 배열의 두 원소 중 큰 값을 반환. fmax는 NaN 무시
minimum, fmin	각 배열의 두 원소 중 작은 값을 반환. fmin는 NaN 무시
mod	첫번째 배열의 원소를 두번째 배열의 원소로 나눈 나머지
copysign	첫번째 배열의 원소의 기호를 두번째 배열의 원소의 기호로 바꿈
greater, greater_equal, less, less_equal, less, less_equal,	각 두 원소 간의 비교 연산(<, <=, >, >=, ==, !=)를 불리언 배열로 반환

유니버설 함수 (ufunc)

- 이항 유니버설 함수

```
arr1 = np.arange(8).reshape(2, -1)
arr2 = np.arange(-40, 40, 10).reshape(2, -1)
print(arr1)
print(arr2)
```

```
[[0 1 2 3]
 [4 5 6 7]]
 [[-40 -30 -20 -10]
 [ 0 10 20 30]]
```

```
np.maximum(arr1, arr2)
```

```
array([[ 0,  1,  2,  3],
 [ 4, 10, 20, 30]])
```

```
np.subtract(arr1, arr2)
```

```
array([[ 40,  31,  22,  13],
 [ 4, -5, -14, -23]])
```

```
np.subtract(arr2, arr1)
```

```
array([[ -40, -31, -22, -13],
 [-4,  5, 14, 23]])
```

```
np.multiply(arr1, arr2)
```

```
array([[ 0, -30, -40, -30],
 [ 0, 50, 120, 210]])
```

기초 통계 메소드

- 기초 통계 메소드
 - 배열 전체 혹은 배열에서 한 축을 따르는 자료에 대한 통계를 계산하는 함수
 - sum
 - mean
 - std, var
 - min, max
 - argmin, argmax
 - cumsum
 - cumprod

```
arr = np.random.randn(200, 500)  
arr.shape
```

```
(200, 500)
```

```
arr.sum()      # ndarray의 메서드 이용
```

```
410.1948188607447
```

```
np.sum(arr)    # numpy의 최상위 함수를 이용
```

```
410.1948188607447
```

기초 통계 메소드

- 기초 통계 메소드
 - sum, mean 등의 함수는 axis를 인자로 받아서 해당 axis에 대한 통계를 계산 가능
 - 축을 지정하지 않을 경우 배열 전체에 대한 값을 연산

<pre>arr.shape</pre>	<p>axis = 0</p> <p>(200, 500)</p>
<pre>arr.mean(axis=0).shape</pre>	<p>(500,)</p>
<pre>arr.mean(axis=1).shape</pre>	<p>(200,)</p>

arr.shape
(200, 500)
arr.mean(axis=0).shape
(500,)
arr.mean(axis=1).shape
(200,)

axis = 0

axis = 1

1 2 3
4 5 6

1 2 3
4 5 6

연습문제 4

- Q1 난수 배열 생성 및 기초 통계치 확인
- Q2 브로드캐스팅
- Q3 mean함수 활용

기타 메소드

- any, all
 - any → 하나 이상의 값이 True이면 True 반환
 - all → 모든 값이 True이면 True 반환

```
arr = np.array([True, False, True])
arr.any()
```

True

```
arr = np.array([True, False, True])
arr.all()
```

False

```
arr = np.array([0, 0, 3])
arr.any()
```

True

```
arr = np.array([0, 0, 1])
arr.all()
```

False

기타 메소드

- any, all
 - any → 하나 이상의 값이 True이면 True 반환
 - all → 모든 값이 True이면 True 반환

```
arr = np.array([1, 1, 1])
arr.all()
```

True

```
arr = np.array([5, -1, np.inf])
arr.all()
```

True

기타 메소드

- where → **np.where(조건, x, y)**
 - x if 조건 else y의 벡터화 버전
 - numpy를 사용하여 큰 배열을 빠르게 처리 가능하며, 다차원 배열도 간결하게 표현 가능

```
xarr = np.array([100, 200, 300, 400])
yarr = np.array([1, 2, 3, 4])
cond = np.array([True, False, True, False])

result = np.where(cond, xarr, yarr)
result

array([100, 2, 300, 4])
```

기타 메소드

- where → np.where(조건, x, y)
 - x와 y자리에 들어가는 인자는 배열이 아니어도 가능

```
xarr = np.array([100, 200, 300, 400])
yarr = np.array([1, 2, 3, 4])
cond = np.array([True, False, True, False])
```

```
np.where(xarr>200, max(xarr), 0)
```

```
array([ 0,  0, 400, 400])
```

```
np.where(xarr%3==0, 1, 0 )
```

```
array([0, 0, 1, 0])
```

기타 메소드

- sort → **arr.sort()**

- 주어진 축에 따라 정렬하며, 다양한 정렬방법들을 지원
- arr 자체를 정렬함 (in-place)

```
np.random.seed(10)
arr = np.random.randint(1, 100, size=10)
arr
array([10, 16, 65, 29, 90, 94, 30, 9, 74, 1])

arr.sort()
arr
array([ 1,  9, 10, 16, 29, 30, 65, 74, 90, 94])
```

기타 메소드

- sort → np.sort(arr, axis=-1)
 - np.sort는 배열을 직접 변경하지 않고 정렬된 결과를 가진 복사본을 반환

```
np.random.seed(20)
arr = np.random.randint(1, 100, size=10)
np.sort(arr)

array([10, 16, 21, 23, 29, 72, 76, 91, 91, 96])

arr

array([91, 16, 96, 29, 91, 10, 21, 76, 23, 72])

-np.sort(-arr)

array([96, 91, 91, 76, 72, 29, 23, 21, 16, 10])
```

선형대수

- 선형대수 연산
 - numpy는 행렬의 곱셈, 분할, 행렬식과 같은 선형대수에 관한 함수들을 제공

Function	설명
dot	내적
dialog	정사각 행렬의 대각/비대각 원소를 1차원 배열로 반환하거나, 1차원 배열을 대각선 원소로 하고 나머지는 0으로 채운 단위행렬 반환
trace	행렬의 대각선 원소의 합을 계산
linalg.det	행렬식을 계산($ad-bc$)
linalg.eig	정사각행렬의 고유값, 고유벡터를 계산
linalg.inv	정사각행렬의 역행렬을 계산
linalg.solve	A 가 정사각 행렬일 때, $Ax = b$ 를 만족하는 x 를 구함
linalg.svd	특이값 분해(SVD)를 계산

선형대수

- dot → np.dot(a, b)
 - dot product: 벡터의 내적

```
x = np.random.randint(-5, 5, size=10)
y = np.random.randint(-10, 10, size=10)
x.shape, y.shape
```

```
((10,), (10,))
```

```
x.dot(y)
```

```
-3
```

```
np.dot(x, y)
```

```
-3
```

$$\mathbf{a} = (a_1, a_2, \dots, a_n)$$

$$\mathbf{b} = (b_1, b_2, \dots, b_n)$$

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

선형대수

- dot → np.dot(a, b)
 - dot product: 벡터의 내적

```
x = np.array([[1, 2, 3], [4, 5, 6]])
y = np.array([[2, -3], [1, 6], [-1, 2]])
x.shape, y.shape
```

```
((2, 3), (3, 2))
```

```
np.dot(x, y)
```

```
array([[ 1, 15],
       [ 7, 30]])
```

선형대수

- matmul → np.matmul(a,b)
 - matrix multiplication: 행렬의 곱

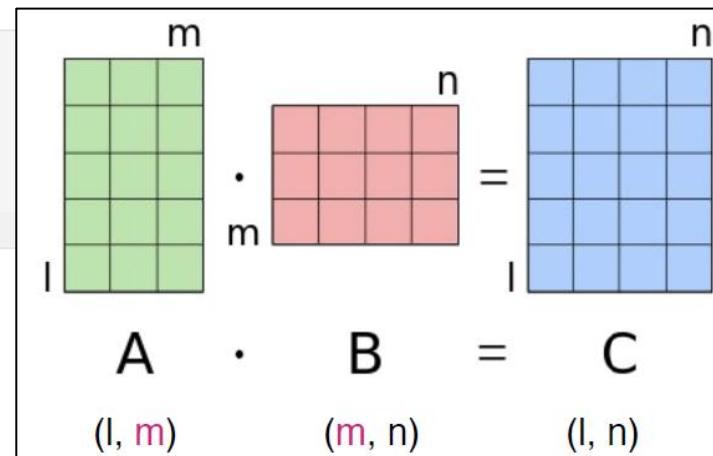
```
a = np.random.randint(-3, 3, 10).reshape(2, 5)
b = np.random.randint(0, 5, 15).reshape(5, 3)
a.shape, b.shape
```

```
((2, 5), (5, 3))
```

```
ab = np.matmul(a, b)
print(ab.shape, '\n')
print(ab)
```

```
(2, 3)
```

```
[[ 6   0   6]
 [-14 -6  -4]]
```



선형대수

- matmul → np.matmul(a,b)
 - matmul을 수행할 시 shape을 유의

```
a = np.random.randint(-3, 3, 10).reshape(2, 5)
b = np.random.randint(0, 5, 15).reshape(5, 3)
a.shape, b.shape

((2, 5), (5, 3))
```

```
np.matmul(b, a)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-75-af3b88aa2232> in <module>()
----> 1 np.matmul(b, a)
```

```
ValueError: shapes (5,3) and (2,5) not aligned: 3 (dim 1) != 2
(dim 0)
```

선형대수

- matmul → np.matmul(a,b)

- 3차원 이상의 경우 마지막 2개 축으로 이루어진 행렬을 다른 축들에 따라 쌓은 것으로 파악
- 즉, 마지막 2개의 차원이 행렬곱이 가능하다면 matmul 가능

```
c = np.arange(24).reshape(2, 3, 4)
d = np.arange(2*4*5).reshape(2, 4, 5)
c.shape, d.shape
((2, 3, 4), (2, 4, 5))

arr = np.matmul(c, d)
arr
array([[[ 70,   76,   82,   88,   94],
       [190,   212,   234,   256,   278],
       [310,   348,   386,   424,   462]],

      [[1510,  1564,  1618,  1672,  1726],
       [1950,  2020,  2090,  2160,  2230],
       [2390,  2476,  2562,  2648,  2734]]])

arr.shape
(2, 3, 5)
```

Practice 5

- Q1 where 활용

주어진 배열을 이용하여 다음과 같은 연산을 해보자.

원소의 값이 0보다 클 경우 배열의 가장 큰 값으로, 0보다 작거나 같을 경우 배열의 가장 작은 값으로 만들어보자.

```
[ ] 1 np.random.seed(253)
    2 arr = np.random.randn(20).reshape(4,5)
```

```
[ ] 1 # 정답을 작성해주세요.
```

```
array([[ 3.30666723, -1.38283849,  3.30666723,  3.30666723,  3.30666723],
       [-1.38283849, -1.38283849, -1.38283849, -1.38283849, -1.38283849],
       [-1.38283849,  3.30666723,  3.30666723,  3.30666723,  3.30666723],
       [-1.38283849, -1.38283849,  3.30666723,  3.30666723,  3.30666723]])
```

Practice 5

- Q2 all 활용

주어진 배열에서 0이 포함된 컬럼이 있는지 없는지 파악해보자.

- 0이 있는 컬럼이 있다면 True
- 없다면 False

```
[ ] 1 np.random.seed(2531)
2 arr = np.random.randint(0,3, (3,4))
3 arr
```

```
array([[2, 2, 0, 0],
       [2, 2, 1, 2],
       [0, 1, 0, 2]])
```

```
[ ] 1 # 정답을 작성해주세요.
```

```
True
```

4. Pandas Basic

판다스 (pandas)

- pandas
 - 데이터 처리, 분석용 라이브러리
 - 표 형식 데이터, 시계열 데이터 등 다양한 형태의 데이터를 다루는데 초점 (CSV, text files, Excel, SQL database)
 - numpy의 배열 기반 계산 스타일을 차용
- pandas의 특징
 - missing data 처리가 용이
 - 축의 이름에 따라 데이터를 정렬할 수 있는 자료구조 제공
 - 일반 데이터베이스처럼 데이터를 합치고 관계연산을 수행하는 기능

Series와 DataFrame

- dataframe
 - R의 dataframe 형식을 사용가능

series

	Date	kospi	kosdaq	gold_fut_132030	Bond_273130
0	2020. 1. 2 오후 3:30:00	2175.17	674.02	10845	108215
1	2020. 1. 3 오후 3:30:00	2176.46	669.93	11000	108565
2	2020. 1. 6 오후 3:30:00	2155.07	655.31	11245	108745
3	2020. 1. 7 오후 3:30:00	2175.54	663.44	11180	108400
4	2020. 1. 8 오후 3:30:00	2151.31	640.94	11360	108270
5	2020. 1. 9 오후 3:30:00	2186.45	666.09	11055	107980
6	2020. 1. 10 오후 3:30:00	2206.39	673.03	11035	107760
7	2020. 1. 13 오후 3:30:00	2229.26	679.22	11080	107695
8	2020. 1. 14 오후 3:30:00	2238.88	678.71	10975	107860

DataFrame

Series와 DataFrame

- Series 생성 → `pd.Series(data, index, dtype, name)`
 - 시리즈는 1차원 배열 같은 자료구조
 - list, numpy, dictionary 등을 이용하여 생성 가능
 - index: 각 row의 이름, 기본적으로는 숫자로 되어 있음
 - dtype: numpy의 dtype과 동일
 - name: 각 column의 이름, 기본값이 없음

Series와 DataFrame

- Series 생성

```
1 obj1 = pd.Series([0, 10, 20, 30, 40, 50, 60])
```

```
2 obj1
```

```
0    0  
1   10  
2   20  
3   30  
4   40  
5   50  
6   60  
dtype: int64
```

```
1 # dtype
```

```
2 obj1 = pd.Series([0, 10, 20, 30, 40, 50, 60], dtype = np.float32)  
3 obj1
```

```
0    0.0  
1   10.0  
2   20.0  
3   30.0  
4   40.0  
5   50.0  
6   60.0  
dtype: float32
```

```
1 # index
```

```
2 obj2 = pd.Series(['a', 'b', 'c'], index = ['i1', 'i2', 'i3'])  
3 obj2
```

```
i1    a  
i2    b  
i3    c  
dtype: object
```

```
1 # dictionary  
2 dic = {'서울특별시' : '02', '경기도' : '031', '인천광역시' : '032', '강원도' : '033',  
3   '충청남도' : '041', '대전광역시' : '042', '충청북도' : '043', '세종특별자치시' : '044'  
4   '부산광역시' : '051', '울산광역시' : '052', '대구광역시' : '053', '경상북도' : '054'  
5   '경상남도' : '055', '전라남도' : '061', '광주광역시' : '062', '전라북도' : '063',  
6   '제주특별자치도' : '064'}
```

```
1 obj3 = pd.Series(dic)
```

```
2 obj3
```

```
서울특별시      02  
경기도          031  
인천광역시      032  
강원도          033  
충청남도        041  
대전광역시      042  
충청북도        043  
세종특별자치시  044  
부산광역시      051  
울산광역시      052  
대구광역시      053  
경상북도        054  
경상남도        055  
전라남도        061  
광주광역시      062  
전라북도        063  
제주특별자치도  064  
dtype: object
```

```
1 # numpy
```

```
2 obj4 = pd.Series(np.arange(4), name='kk')  
3 obj4
```

```
0    0  
1    1  
2    2  
3    3  
Name: kk, dtype: int64
```

Series와 DataFrame

- Series의 index, values, dtype

```
1 obj1  
0    0.0  
1   10.0  
2   20.0  
3   30.0  
4   40.0  
5   50.0  
6   60.0  
dtype: float32
```

```
1 obj1.index
```

```
RangeIndex(start=0, stop=7, step=1)
```

```
1 obj1.values
```

```
array([ 0., 10., 20., 30., 40., 50., 60.], dtype=float32)
```

```
1 obj1.dtype
```

```
dtype('float32')
```

```
1 obj2  
i1    a  
i2    b  
i3    c  
dtype: object
```

```
1 obj2.index
```

```
Index(['i1', 'i2', 'i3'], dtype='object')
```

```
1 obj2.values
```

```
array(['a', 'b', 'c'], dtype=object)
```

Series와 DataFrame

- Series의 index, values, dtype

```
1 obj3
```

```
서울특별시      02
경기도        031
인천광역시      032
강원도        033
충청남도      041
대전광역시      042
충청북도      043
세종특별자치시  044
부산광역시      051
울산광역시      052
대구광역시      053
경상북도      054
경상남도      055
전라남도        061
광주광역시      062
전라북도        063
제주특별자치도  064
dtype: object
```

```
1 obj3.index
```

```
Index(['서울특별시', '경기도', '인천광역시', '강원도', '충청남도', '대전광역시', '충청북도', '세종특별자치시',
       '부산광역시', '울산광역시', '대구광역시', '경상북도', '경상남도', '전라남도', '광주광역시', '전라
       '제주특별자치도'],
      dtype='object')
```

```
1 obj3.values
```

```
array(['02', '031', '032', '033', '041', '042', '043', '044', '051',
       '052', '053', '054', '055', '061', '062', '063', '064'],
      dtype=object)
```

```
1 obj3.dtype
```

```
dtype('O')
```

```
1 obj4
```

```
0    0
1    1
2    2
3    3
```

```
Name: kk, dtype: int64
```

```
1 obj4.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

```
1 obj4.values
```

```
array([0, 1, 2, 3])
```

```
1 obj4.dtype
```

```
dtype('int64')
```

Series와 DataFrame

- DataFrame 생성
→ `pd.DataFrame(data, index, dtype, columns)`
 - dictionary나 numpy 배열을 이용하여 생성
 - index: 각 row의 이름, 기본적으로 숫자로 되어 있음
 - dtype: numpy의 dtype과 동일
 - columns: 각 columns의 이름, 기본값이 없음

Series와 DataFrame

- Data Frame 생성

```
[ ] 1 #dictionary
2 data = {'제목': ['극한직업', '어벤져스: 엔드게임', '알라딘', '기생충', '엑시트', '스파이더맨: 파 프
3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
6 frame1 = pd.DataFrame(data)
7 frame1
```

	제목	감독	관객수
0	극한직업	이병헌	16264944
1	어벤져스: 엔드게임	안소니 루소,조 루소	13934592
2	알라딘	가이 리치	12551956
3	기생충	봉준호	10084564
4	엑시트	이상근	9424431
5	스파이더맨: 파 프롬 홈	존 왓츠	8020208
6	겨울왕국 2	크리스 벅,제니퍼 리	7603956
7	캡틴 마블	애너 보든,라이언 플렉	5801069
8	조커	토드 필립스	5245841
9	봉오동 전투	원신연,이영용	4777456

Series와 DataFrame

- Data Frame 생성

```
1 #dictionary
2 data = {'1월': {'식비': 300000, '교통비' : 50000, '기타' : 7000},
3     |   '2월': {'식비': 621000, '교통비' : 75900, '기타' : 9000},
4     |   '3월': {'식비': 238400, '교통비' : 28900, '기타' : 1500},}
5 frame2 = pd.DataFrame(data)
6 frame2
```

	1월	2월	3월
식비	300000	621000	238400
교통비	50000	75900	28900
기타	7000	9000	1500

```
1 # numpy
2 frame3 = pd.DataFrame(np.arange(30).reshape(6,-1))
3 frame3
```

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24
5	25	26	27	28	29

Series와 DataFrame

- Data Frame의 index, values, dtype

```
1 frame1.index  
RangeIndex(start=0, stop=10, step=1)  
  
1 frame1.columns  
Index(['제목', '감독', '관객수'], dtype='object')  
  
1 frame1.values  
array([['극한직업', '이병헌', 16264944],  
       ['어벤져스: 엔드게임', '안소니 루소, 조 루소', 13934592],  
       ['알라딘', '가이 리치', 12551956],  
       ['기생충', '봉준호', 10084564],  
       ['엑시트', '이상근', 9424431],  
       ['스파이더맨: 파 프롬 흠', '존 왓츠', 8020208],  
       ['겨울왕국 2', '크리스 벅, 제니퍼 리', 7603956],  
       ['캡틴 마블', '애너 보든, 라이언 플렉', 5801069],  
       ['조커', '토드 필립스', 5245841],  
       ['봉오동 전투', '원신연, 이영용', 4777456]], dtype=object)
```

Series와 DataFrame

- Data Frame의 index, values, dtype

```
1 frame3.index  
RangeIndex(start=0, stop=6, step=1)  
  
1 frame3.columns  
RangeIndex(start=0, stop=5, step=1)  
  
1 frame3.values  
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19],  
       [20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29]])
```

Data Processing

- 데이터 로딩

pandas file 파일 파싱 함수	설명
read_csv	파일, url, 파일과 유사한 객체로부터 구분된 데이터 읽어옴. 데이터 구분자는 쉼표(,)가 기본
read_excel	엑셀(xls, xlsx)에서 표 형식의 데이터를 읽어옴
read_json	JSON 문자열에서 데이터를 읽어옴
read_pickle	파이썬 피클 포맷으로 저장된 객체를 읽어옴

- 데이터 저장

pandas file 저장 함수	설명
to_csv	csv 형식으로 저장
to_json	JSON 형식으로 저장
to_pickle	파이썬 피클 포맷으로 저장

Data Processing

- `read_csv()`, `read_json()`

```
1 # 데이터 읽기
2 stock = pd.read_csv('stock_2020_01.csv',
3 | | | | | encoding = "euc-kr")
```

1 stock					
	Date	kospi	kosdaq	gold_fut_132030	Bond_273130
0	2020. 1. 2 오후 3:30:00	2175.17	674.02	10845	108215
1	2020. 1. 3 오후 3:30:00	2176.46	669.93	11000	108565
2	2020. 1. 6 오후 3:30:00	2155.07	655.31	11245	108745
3	2020. 1. 7 오후 3:30:00	2175.54	663.44	11180	108400
4	2020. 1. 8 오후 3:30:00	2151.31	640.94	11360	108270
5	2020. 1. 9 오후 3:30:00	2186.45	666.09	11055	107980
6	2020. 1. 10 오후 3:30:00	2206.39	673.03	11035	107760
7	2020. 1. 13 오후 3:30:00	2229.26	679.22	11080	107695
8	2020. 1. 14 오후 3:30:00	2238.88	678.71	10975	107860
9	2020. 1. 15 오후 3:30:00	2230.98	679.16	11060	108020
10	2020. 1. 16 오후 3:30:00	2248.05	686.52	11060	108015
11	2020. 1. 17 오후 3:30:00	2250.57	688.41	11075	107830
12	2020. 1. 20 오후 3:30:00	2262.64	683.47	11140	107775
13	2020. 1. 21 오후 3:30:00	2239.69	676.52	11155	108040
14	2020. 1. 22 오후 3:30:00	2267.25	688.25	11050	107995
15	2020. 1. 23 오후 3:30:00	2246.13	685.57	11080	107970
16	2020. 1. 28 오후 3:30:00	2176.72	664.70	11250	108450
17	2020. 1. 29 오후 3:30:00	2185.28	670.18	11160	108585
18	2020. 1. 30 오후 3:30:00	2148.00	656.39	11225	108810
19	2020. 1. 31 오후 3:30:00	2119.01	642.48	11210	108795

```
1 movie_2019 = pd.read_json('movie_2019.json')
```

순위	영화명	개봉일	매출액 액 점 유율	관객수	스크 린수	상영횟수	대표국적	국적	배급사
0	1 극한직업	2019-01-23	139651845516	0.073	16265618	2003	292584	한국	한국 씨제이이엔엠(주)
1	2 어벤져스: 엔드게임	2019-04-24	122182694160	0.064	13934592	2835	242001	미국	미국 월트디즈니컴퍼니코리아 유한책임회사
2	3 겨울왕국 2	2019-11-21	111596248720	0.058	13369064	2648	282557	미국	미국 월트디즈니컴퍼니코리아 유한책임회사
3	4 알라딘	2019-05-23	106955138359	0.056	12552283	1409	266469	미국	미국 월트디즈니컴퍼니코리아 유한책임회사
4	5 기생충	2019-05-30	85883963645	0.045	10085275	1948	192855	한국	한국 씨제이이엔엠(주)
5	6 엑시트	2019-07-31	79232012162	0.041	9426011	1660	202223	한국	한국 씨제이이엔엠(주)
6	7 스파이더맨: 파프롬홈	2019-07-02	69010000100	0.036	8021145	2142	180474	미국	미국 소니피쳐스엔터테인먼트코리아 주식회사극장배급지점
7	8 백두산	2019-12-19	52905807770	0.028	6290504	1971	99915	한국	한국 씨제이이엔엠(주)(주)덱스터스튜디오
8	9 캡틴 마블	2019-03-06	51507488723	0.027	5802810	2100	186382	미국	미국 월트디즈니컴퍼니코리아 유한책임회사
9	10 조커	2019-10-02	45381075450	0.024	5247874	1418	147380	미국	미국 워너브러더스 코리아(주)
10	11 봉오동 전투	2019-08-07	40588648538	0.021	4787538	1476	107030	한국	한국 (주)쇼박스

indexing & slicing

- Series
 - integer-location based
 - numpy indexing과 유사하게 동작
 - label-location based
 - label(index)를 기준으로 동작
 - Series.iloc
 - integer-location based property
 - Series.loc
 - label-location based property

indexing & slicing

- Series

- integer-location based
 - numpy indexing과 유사하게 동작
 - 리스트에 위치(int)를 넘겨주면 그 위치에 해당되는 자료들을 index(label) 값과 함께 반환

```
obj = pd.Series(np.arange(8), index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
obj
obj[3]
obj[-1]
obj[[1,3,5]]
```

The screenshot shows a Jupyter Notebook cell with the following code:

```
obj = pd.Series(np.arange(8), index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
obj
obj[3]
obj[-1]
obj[[1,3,5]]
```

The output pane displays the results of each command:

- `obj`: A Series object with 8 integer values from 0 to 7, indexed by letters 'a' through 'h'.

Index	Value
a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
- `obj[3]`: Returns the value at index 3, which is 3.
- `obj[-1]`: Returns the value at index -1, which is 7.
- `obj[[1,3,5]]`: Returns a subset Series containing elements at indices 1, 3, and 5, resulting in values [1, 3, 5].

indexing & slicing

- Series
 - integer-location based
 - numpy slicing과 유사하게 동작
 - numpy boolean indexing과 유사하게 동작

```
obj = pd.Series(np.arange(8), index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
obj
obj[1:3]
# boolean
obj[obj<3]
```

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
	dtype: int64

b	1
c	2
	dtype: int64

a	0
b	1
c	2
	dtype: int64

indexing & slicing

- Series
 - label-location based
 - label(index)를 기준으로 동작

```
obj['c']
```

```
2
```

```
obj.c
```

```
2
```

```
obj[['e', 'c']] → 리스트에 담아서 여러 개의 자료를 indexing
```

```
e    4  
c    2  
dtype: int64
```

indexing & slicing

- Series
 - label-location based
 - label(index)를 기준으로 동작

주의: 라벨로 슬라이싱하면 시작점과 끝점을 포함!!!

```
obj['a':'c']
```

```
a    0  
b    1  
c    2  
dtype: int64
```

indexing & slicing

- Series
 - label-location based
 - slicing을 이용해 값을 할당 가능

```
obj['d':'e']=100
```

```
obj
```

```
a      0  
b      1  
c      2  
d    100  
e    100  
f      5  
g      6  
h      7  
dtype: int64
```

indexing & slicing

- Series
 - Series.iloc
 - integer-location based property

```
obj.iloc[2]
```

```
2
```

```
obj.iloc[[2]]
```

```
c    2  
dtype: int64
```

```
obj.iloc[1:4]
```

```
b    1  
c    2  
d    3  
dtype: int64
```

indexing & slicing

- Series
 - Series.loc
 - label-location based property

```
obj.loc['a']  
0  
  
obj.loc[['a']]  
a    0  
dtype: int64  
  
obj.loc[['a', 'c']]  
a    0  
c    2  
dtype: int64  
  
obj.loc['a':'c'] → label로 슬라이싱을 하기 때문에 끝 데이터 포함  
a    0  
b    1  
c    2  
dtype: int64
```

indexing & slicing

- DataFrame
 - indexing
 - 컬럼을 기준으로 동작

```
frame = pd.DataFrame(np.arange(24).reshape(4,-1),
                     columns = ['c1', 'c2', 'c3', 'c4', 'c5'],
                     index = ['r1', 'r2', 'r3', 'r4'])
```

frame

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

frame['c3']	frame.c3
r1 2	r1 2
r2 8	r2 8
r3 14	r3 14
r4 20	r4 20
Name: c3, dtype: int64	Name: c3, dtype: int64

indexing & slicing

- DataFrame

- indexing

- 컬럼을 기준으로 동작

```
frame = pd.DataFrame(np.arange(24).reshape(4,-1),
                     columns = ['c1', 'c2', 'c3', 'c4', 'c5'],
                     index = ['r1', 'r2', 'r3', 'r4'])

frame
```

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

frame[['c1', 'c2']]

	c1	c2
r1	0	1
r2	6	7
r3	12	13
r4	18	19

indexing & slicing

- DataFrame
 - slicing
 - 인덱스(label)을 기준으로 동작

```
frame = pd.DataFrame(np.arange(24).reshape(4,-1),  
                      columns = ['c1', 'c2', 'c3', 'c4', 'c5',  
                      index = ['r1', 'r2', 'r3', 'r4'])
```

```
frame
```

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

```
frame['r1':'r2']
```

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11

→ 컬럼 기준
슬라이싱의
결과에는 값이
없음

```
frame['c1':'c2']
```

	c1	c2	c3	c4	c5	c6

indexing & slicing

- DataFrame

- DataFrame.iloc → **df.iloc(row, column)**
 - integer-location based property

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

```
frame.iloc[[0], [3]]
```

```
c4  
---  
r1  3
```

```
frame.iloc[[0,1], 1:4]
```

```
c2  c3  c4  
---  
r1  1   2   3  
r2  7   8   9
```

indexing & slicing

- DataFrame

- DataFrame.loc → `df.loc(row, column)`
 - label-location based property

	c1	c2	c3	c4	c5	c6
r1	0	1	2	3	4	5
r2	6	7	8	9	10	11
r3	12	13	14	15	16	17
r4	18	19	20	21	22	23

```
frame.loc[['r1'], ['c4']]
```

```
c4  
---  
r1 3
```

```
frame.loc['r1':'r2', ['c2', 'c3', 'c4']]
```

```
c2  c3  c4  
---  
r1  1   2   3  
r2  7   8   9
```

Practice 6

- Q1 Series 활용
- Q2 DataFrame 활용

```
series = pd.Series(np.arange(5), index=list('abcde'))  
series
```

```
a    0  
b    1  
c    2  
d    3  
e    4  
dtype: int64
```

```
frame = pd.DataFrame(np.arange(16).reshape(4,-1),  
                     columns = ['c1', 'c2', 'c3', 'c4'],  
                     index = ['r1', 'r2', 'r3', 'r4'])  
frame
```

	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

Quiz 1

- Pandas Basic Quiz!

주어진 리스트를 newspaper라는 변수명의 Series 객체로 만들기

```
s =['경향신문신문', '국민일보신문', '동아일보신문', '문화일보신문', '서울  
'세계일보신문', '조선일보신문', '중앙일보신문', '한겨레신문', '한국일  
'''  
output:  
0    경향신문신문  
1    국민일보신문  
2    동아일보신문  
3    문화일보신문  
4    서울신문신문  
5    세계일보신문  
6    조선일보신문  
7    중앙일보신문  
8    한겨레신문  
9    한국일보신문  
dtype: object  
'''
```

산술연산

- 산술연산
 - 산술 연산자 혹은 산술 연산 메소드를 사용하여 연산

메소드	설명
add, radd	덧셈(+)을 위한 메소드
sub, rsub	뺄셈(-)을 위한 메소드
mul, rmul	곱셈(*)을 위한 메소드
div, rdiv	나눗셈(/)을 위한 메소드
pow, rpow	거듭제곱(**)을 위한 메소드
floordiv, rfloordiv	소수점 내림(/)을 위한 메소드

산술연산

- Series
 - index를 기준으로 연산

```
s1 = pd.Series([1, 2, 3, 4], index = ['a', 'b', 'c', 'd'])

s2 = pd.Series([10, 20, 30, 40], index = ['a', 'b', 'c', 'd'])

s1+s2

a    11
b    22
c    33
d    44
dtype: int64
```

산술연산

- Series

- 결과 값에 두 인덱스의 값이 통합됨
- 짹이 맞지 않는 인덱스가 있는 경우, outer join과 유사하게 동작
→ NaN 으로 반환

```
s1 = pd.Series([1, 2, 3, 4], index = ['a', 'b', 'c', 'd'])
```

```
s3 = pd.Series([2, 2, 2, 2], index = ['b', 'c', 'd', 'e'])
```

```
# a, e  
s1+s3
```

```
a      NaN  
b      4.0  
c      5.0  
d      6.0  
e      NaN  
dtype: float64
```

산술연산

- Series
 - 결측치의 기본값 지정 가능 → fill_value 활용

```
s1.add(s3, fill_value=0)
```

```
a    1.0
b    4.0
c    5.0
d    6.0
e    2.0
dtype: float64
```

산술연산

- DataFrame
 - index, column을 기준으로 연산

	c1	c2
id001	100.0	100.0
id002	100.0	100.0
id003	100.0	100.0

	c2	c3
id002	0	200
id003	400	600
id004	800	1000

d1 + d2

	c1	c2	c3
id001	NaN	NaN	NaN
id002	NaN	100.0	NaN
id003	NaN	500.0	NaN
id004	NaN	NaN	NaN

산술연산

- DataFrame
 - 두 df에 모두 없는 경우 NaN으로 채워짐

	c1	c2
id001	100.0	100.0
id002	100.0	100.0
id003	100.0	100.0

	c2	c3
id002	0	200
id003	400	600
id004	800	1000

```
# (id001, c3), (id004, c1)  
d1.add(d2, fill_value=0)
```

	c1	c2	c3
id001	100.0	100.0	NaN
id002	100.0	100.0	200.0
id003	100.0	500.0	600.0
id004	NaN	800.0	1000.0

산술연산

- Series와 DataFrame 간의 연산
 - Series의 인덱스(label)을 DataFrame의 컬럼에 맞추고 row로 전파

frame			
	a	b	c
0	1	1	1
1	1	1	1
2	1	1	1

series	
a	100
b	200
c	300
dtype: int64	

frame + series			
	a	b	c
0	101	201	301
1	101	201	301
2	101	201	301

산술연산

- Series와 DataFrame 간의 연산
 - axis를 인자로 축을 지정하여 column으로 전파 가능

frame			
	a	b	c
0	1	1	1
1	1	1	1
2	1	1	1

series2	
0	100
1	200
2	300
	dtype: int64

```
frame.add(series2, axis=0)
```

	a	b	c
0	101	101	101
1	201	201	201
2	301	301	301

5. Advanced Pandas

기술통계

- 요약통계 메소드

메서드	설명
head, tail	series나 dataframe의 몇 개 데이터만 보여줌
describe	series나 dataframe의 각 컬럼에 대한 요약 통계
count	na 값을 제외한 값의 개수를 반환
min, max	최소값, 최대값
argmin, argmax	최소값, 최대값을 가진 색인의 위치(정수)를 반환
idxmin, idxmax	최소값, 최대값을 가진 색인의 값을 반환
sum	합
mean, median	평균, 중앙값
var, std	분산, 표준편차

기술통계

- head / tail
 - head: 위에서부터 일정한 개수만큼의 데이터만 제시
 - tail: 아래에서부터 일정한 개수만큼의 데이터 제시

```
stock.head()
```

	Date	kospi	kosdaq	gold_fut_132030	Bond_273130
0	2020. 1. 2 오후 3:30:00	2175.17	674.02	10845	108215
1	2020. 1. 3 오후 3:30:00	2176.46	669.93	11000	108565
2	2020. 1. 6 오후 3:30:00	2155.07	655.31	11245	108745
3	2020. 1. 7 오후 3:30:00	2175.54	663.44	11180	108400
4	2020. 1. 8 오후 3:30:00	2151.31	640.94	11360	108270

기술통계

- **describe**

- 평균, 표준편차, 최소/최대값, 사분위수 제공
- 데이터 타입이 섞여 있으면 수치형 데이터로 이루어진 컬럼만 요약

The screenshot shows a Jupyter Notebook interface. On the left, code is written to create a DataFrame named 'frame'. On the right, the output of the 'frame.describe()' command is shown as a table.

Code:

```
frame = pd.DataFrame([[10, 4, 'a'], [10, 2.5, 'b'],
                      index=['i1', 'i2', 'i3'],
                      columns=['c1', 'c2', 'c3'])
```

Output:

	c1	c2
count	2.0	3.000000
mean	10.0	5.500000
std	0.0	3.968627
min	10.0	2.500000
25%	10.0	3.250000
50%	10.0	4.000000
75%	10.0	7.000000
max	10.0	10.000000

기술통계

- `describe`
 - 인자로 `include='object'`를 주면 object에 대한 요약 통계를 제공

The screenshot shows a Jupyter Notebook interface. On the left, a code cell contains:frame = pd.DataFrame([[10, 4, 'a'], [10, 2.5, 'b'], [NaN, 10.0, 'a']], index=['i1', 'i2', 'i3'], columns=['c1', 'c2', 'c3'])
frameOn the right, the output of the code cell is shown. A tooltip for the `frame.describe(include='object')` call is displayed, showing the results for column 'c3':

c3	
count	3
unique	2
top	a
freq	2

기술통계

- **describe**

- 인자로 `include='all'`을 주면 전체에 대한 요약 통계를 제공

```
frame = pd.DataFrame([[10, 4, 'a'], [10,
                                         index=['i1', 'i2',
                                                 columns=['c1', 'c2']
frame
```

	c1	c2	c3
i1	10.0	4.0	a
i2	10.0	2.5	b
i3	NaN	10.0	a

	c1	c2	c3
count	2.0	3.000000	3
unique	NaN	NaN	2
top	NaN	NaN	a
freq	NaN	NaN	2
mean	10.0	5.500000	NaN
std	0.0	3.968627	NaN
min	10.0	2.500000	NaN
25%	10.0	3.250000	NaN
50%	10.0	4.000000	NaN
75%	10.0	7.000000	NaN
max	10.0	10.000000	NaN

기술통계

- max / min / sum
 - 축을 지정하지 않으면 row(0)을 기준으로 연산
 - 컬럼을 기준으로 연산하고 싶으면 axis='columns' 또는 '1' 사용

```
frame.max()
```

```
c1    10  
c2    10  
c3     b  
dtype: object
```

```
frame.min(axis=1)
```

```
i1     4.0  
i2     2.5  
i3    10.0  
dtype: float64
```

```
frame.sum()
```

```
c1     20  
c2    16.5  
c3    aba  
dtype: object
```

기술통계

- `unique`
 - 중복되는 값을 제거하고 유일값만 담고 있는 Series를 반환

```
obj = pd.Series([2, 1, 3, 3, 1, 5, np.nan, 1, 2])
```

```
obj
```

```
0    2.0  
1    1.0  
2    3.0  
3    3.0  
4    1.0  
5    5.0  
6    NaN  
7    1.0  
8    2.0  
dtype: float64
```

```
obj.unique()
```

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

기술통계

- `value_counts`

- 값을 인덱스(label)로 하고 그 값의 개수를 담고 있는 Series를 반환

```
obj.value_counts()
```

```
1.0      3  
3.0      2  
2.0      2  
5.0      1  
dtype: int64
```

```
# normalize  
obj.value_counts(normalize=True)
```

```
1.0      0.375  
3.0      0.250  
2.0      0.250  
5.0      0.125  
dtype: float64
```

정렬

- Series.sort_index

- index를 기준으로 정렬

→ `series.sort_index(axis=0, ascending=True, na_position='last')`

- axis: 정렬 기준 축으로 series에서는 0만 가능
- ascending: True이면 오름차순, False이면 내림차순으로 정렬
- na_position: 정렬시 NaN값의 위치 지정, {'first', 'last'}

obj	
a	1
d	2
e	3
b	-1
c	-2
dtype: int64	

obj.sort_index()	
a	1
b	-1
c	-2
d	2
e	3
dtype: int64	

obj.sort_index(ascending=False)	
e	3
d	2
c	-2
b	-1
a	1
dtype: int64	

정렬

- Series.sort_values

- 값을 기준으로 정렬

→`series.sort_values(axis=0, ascending=True, na_position='last')`

- axis: 정렬 기준 축으로 series에서는 0만 가능
- ascending: True이면 오름차순, False이면 내림차순으로 정렬
- na_position: 정렬시 NaN값의 위치 지정, {'first', 'last'}

```
obj  
a    1  
d    2  
e    3  
b   -1  
c   -2  
dtype: int64
```

```
obj.sort_values()  
3    0.0  
0    10.0  
2    20.0  
1    NaN  
4    NaN  
dtype: float64
```

```
obj.sort_values(ascending=False)  
2    20.0  
0    10.0  
3    0.0  
1    NaN  
4    NaN  
dtype: float64
```

```
obj.sort_values(na_position='first')  
1    NaN  
4    NaN  
3    0.0  
0    10.0  
2    20.0  
dtype: float64
```

정렬

- df.sort_index
 - index를 기준으로 정렬
 - `df.sort_index(axis=0, ascending=True, na_position='last')`
 - axis: 정렬 기준 축으로 df에서는 0('index')과 1('column') 가능
 - ascending: True이면 오름차순, False이면 내림차순으로 정렬
 - na_position: 정렬시 NaN값의 위치 지정, {'first', 'last'}

frame			
	e	d	f
a	0	1	2
c	3	4	5
b	6	7	8

frame.sort_index()			
	e	d	f
a	0	1	2
b	6	7	8
c	3	4	5

frame.sort_index(axis=1)			
	d	e	f
a	1	0	2
c	4	3	5
b	7	6	8

정렬

- df.sort_values

- 값을 기준으로 정렬

→`df.sort_values(by, axis=0, ascending=True, na_position='last')`

- by: 여러 개의 컬럼이나 인덱스를 정렬할 때 유용한 인자 (정렬 기준)
- axis: 정렬 기준 축으로 df에서는 0('index')과 1('column') 가능
- ascending: True이면 오름차순, False이면 내림차순으로 정렬
- na_position: 정렬시 NaN값의 위치 지정, {'first', 'last'}

frame			
	e	d	f
a	0	1	2
c	3	4	5
b	6	7	8

frame.sort_values(by='a', ascending = False)		
	a	b
2	10	3
3	10	1
4	8	4
0	5	2
1	4	0

정렬

- df.sort_values

- 값을 기준으로 정렬

→`df.sort_values(by, axis=0, ascending=True, na_position='last')`

- by: 여러 개의 컬럼이나 인덱스를 정렬할 때 유용한 인자 (정렬 기준)
- axis: 정렬 기준 축으로 df에서는 0('index')과 1('column') 가능
- ascending: True이면 오름차순, False이면 내림차순으로 정렬
- na_position: 정렬시 NaN값의 위치 지정, {'first', 'last'}

frame			
	e	d	f
a	0	1	2
c	3	4	5
b	6	7	8

frame.sort_values(by=['a', 'b'], ascending = [False, True])			
	a	b	
3	10	1	
2	10	3	
4	8	4	
0	5	2	
1	4	0	

by에 여러 개의 기준을 주고 싶으면 리스트로 인자 전달, 리스트에 있는 순서가 우선 순위가 되며, 이에 상응하여 ascending 인자도 multi-value를 입력 가능

함수 적용과 맵핑

- series: map → `series.map(arg, na_anction=None)`
 - map은 series의 각각의 element들을 다른 값으로 대체
 - arg: 대체할 값으로 function, dictionary, series 등
 - na_action: NaN값이 존재할 경우 어떻게 작동할지에 대한 인자
{None, 'ignore'} None이라면 NaN에도 작동, 'ignore'는 NaN 무시

```
series
0    100
1    200
2    300
dtype: int64
```

```
series.map('${}'.format)
```

```
0    $100
1    $200
2    $300
dtype: object
```

```
series.map('{}달러'.format)
```

```
0    100달러
1    200달러
2    300달러
dtype: object
```

```
# lambda
f = lambda x: np.add(x, 3)
series.map(f)
```

```
0    103
1    203
2    303
dtype: int64
```

```
series.map({100:'C', 200:'B', 300:'A'})
```

```
0    C
1    B
2    A
dtype: object
```

함수 적용과 맵핑

- series: apply → `series.apply(func, args, **kwds)`
 - 어떠한 함수를 series의 각각의 element에 적용시켜주는 함수
 - map함수보다 적용할 수 있는 함수의 범위가 넓음
 - func: series에 적용될 함수
 - args: series를 제외한 함수에 들어갈 다른 매개변수
 - **kwds: 함수에 넘겨줄 키워드 인자들

```
s  
London    20  
New York  21  
Helsinki  12  
dtype: int64
```

```
def sub_custom_value(x, val) :  
    return x - val  
  
s.apply(sub_custom_value, args=(10,))  
  
London      10  
New York   11  
Helsinki   2  
dtype: int64
```

함수 적용과 맵핑

- series: apply → **series.apply(func, args, **kwds)**
 - 어떠한 함수를 series의 각각의 element에 적용시켜주는 함수
 - map함수보다 적용할 수 있는 함수의 범위가 넓음
 - func: series에 적용될 함수
 - args: series를 제외한 함수에 들어갈 다른 매개변수
 - **kwds: 함수에 넘겨줄 키워드 인자들

```
s  
London    20  
New York  21  
Helsinki 12  
dtype: int64
```

```
def add_custom_values(x, **kwargs):  
    for month in kwargs:  
        x += kwargs[month]  
    return x  
  
s.apply(add_custom_values, june=30 , july=20, august=25)  
London    95  
New York  96  
Helsinki 87  
dtype: int64
```

함수 적용과 맵핑

- df: apply → `df.apply(func, axis)`
 - 어떠한 함수를 series의 각각의 element에 적용시켜주는 함수
 - map함수보다 적용할 수 있는 함수의 범위가 넓음
 - func: df에 적용될 함수
 - axis: 함수에 적용될 기준 축

frame				
	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
frame.apply(lambda x: x.max()-x.min())
```

```
a    8  
b    8  
c    8  
d    8  
dtype: int64
```

```
frame.apply(lambda x: x.max()-x.min(), axis=1)
```

```
0    3  
1    3  
2    3  
dtype: int64
```

함수 적용과 맵핑

- df: apply → `df.apply(func, axis)`
 - 어떠한 함수를 series의 각각의 element에 적용시켜주는 함수
 - map함수보다 적용할 수 있는 함수의 범위가 넓음
 - func: df에 적용될 함수
 - axis: 함수에 적용될 기준 축

frame				
	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

```
frame.apply(lambda x: x.max() - x.min(), axis=1)
0    3
1    3
2    3
dtype: int64

frame.applymap(lambda x: x**2)
      a   b   c   d
0    0   1   4   9
1   16  25  36  49
2   64  81 100 121
```

Quiz 2

- stock 데이터 및 ecocycle 데이터 활용

stock 데이터와 ecocycle 데이터 읽어오기

파일명 : quiz_1_stock.csv

변수명: stock

파일명: quiz_2_ecocycle.csv

변수명: eco

encoding:'euc-kr'

```
stock =
```

```
eco =
```

데이터 정제 및 준비

- Day4 data preprocessing 참조

- 데이터 삭제 - drop

- row나 column에서 특정한 label을 삭제하는 함수
- 기본 axis=0

frame				
	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

frame.drop('r1')				
	c1	c2	c3	c4
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

frame.drop('c2', axis = 1)			
	c1	c3	c4
r1	0	2	3
r2	4	6	7
r3	8	10	11
r4	12	14	15

데이터 정제 및 준비

- 데이터 삭제 - drop

- row나 column에서 특정한 label을 삭제하는 함수
- 리스트로 columns을 지정 가능
- inplace 옵션 및 반환값으로 할당 가능

frame				
	c1	c2	c3	c4
r1	0	1	2	3
r2	4	5	6	7
r3	8	9	10	11
r4	12	13	14	15

frame.drop(columns=['c3', 'c4'])

# inplace	
frame.drop(['r2'], inplace=True)	frame
r1	0 1 2 3
# 할당	
frame = frame.drop(['r3'])	frame
r1	0 1 2 3
r4	12 13 14 15

데이터 정제 및 준비

- 데이터 병합 – concat

→ `pd.concat(objs, axis, join='outer')`

- axis: 기준이 되는 축

- join: 다른 축에 대해 어떠한 방식으로 이어 붙일지 {'outer', 'inner'}

```
print(s1, s2, s3, sep='\n\n')
```

```
c    100  
b    200  
dtype: int64
```

```
c    300  
d    300  
e    300  
dtype: int64
```

```
f    500  
g    600  
dtype: int64
```

```
pd.concat([s1, s2, s3])
```

```
c    100  
b    200  
c    300  
d    300  
e    300  
f    500  
g    600  
dtype: int64
```

데이터 정제 및 준비

- 데이터 병합 – concat

→ `pd.concat(objs, axis, join='outer')`

- axis: 기준이 되는 축

- join: 다른 축에 대해 어떠한 방식으로 이어 붙일지 {'outer', 'inner'}

```
print(s1, s2, s3, sep='\n\n')

c    100
b    200
dtype: int64

c    300
d    300
e    300
dtype: int64

f    500
g    600
dtype: int64
```

```
pd.concat([s1, s2], axis=1)

      0      1
c  100.0  300.0
b  200.0    NaN
d    NaN  300.0
e    NaN  300.0
```

→ 새로운 컬럼 생성

데이터 정제 및 준비

- 데이터 병합 – concat
→ `pd.concat(objs, axis, join='outer')`
 - 리스트의 형식으로 병합 가능

```
data1 = pd.DataFrame({'id': ['01', '02', '03', '04', '05', '06'],
                      'col1': np.random.randint(1, 100, 6),
                      'col2': np.random.randint(1000, 10000, 6)})
data2 = pd.DataFrame({'id': ['04', '05', '06', '07'],
                      'col1': np.random.randint(1, 100, 4),
                      'col2': np.random.randint(1000, 10000, 4)})  
  
pd.concat([data1, data2])
```

	id	col1	col2
0	01	43	1918.0
1	02	17	1103.0
2	03	31	1269.0
3	04	12	1991.0
4	05	19	1853.0
5	06	1	1127.0
0	04	3876	NaN
1	05	2076	NaN
2	06	1509	NaN
3	07	4533	NaN

데이터 정제 및 준비

- 데이터 병합 – concat
→ `pd.concat(objs, axis, join='outer')`
 - 리스트의 형식으로 병합 가능

pd.concat([data1, data2], axis=1)					
	id	col1	col2	id	col1
0	01	43	1918	04	3876.0
1	02	17	1103	05	2076.0
2	03	31	1269	06	1509.0
3	04	12	1991	07	4533.0
4	05	19	1853	NaN	NaN
5	06	1	1127	NaN	NaN

데이터 정제 및 준비

- 데이터 병합 – merge

→ `pd.merge(objs, how='inner', on)`

- objs: 병합할 자료
- how: 어떤 방식으로 병합할 것인가 {'inner', 'outer', 'left', 'right'}
- on: 어떤 label을 기준으로 병합할 것인가, 기본적으로 중복되는 컬럼을 기준으로 병합 진행

data1			
	id	col1	col2
0	01	5	1083
1	02	29	1548
2	03	6	1594
3	04	16	1763
4	05	7	1305
5	06	20	1070

data2		
	id	col1
0	04	4532
1	05	2599
2	06	4556
3	07	3547

#inner join

```
pd.merge(data1, data2, on='id')
```

	id	col1_x	col2	col1_y
0	04	16	1763	4532
1	05	7	1305	2599
2	06	20	1070	4556

데이터 정제 및 준비

- 데이터 병합 – merge
→ `pd.merge(objs, how='inner', on)`
 - 중복되는 키가 없을 경우, `left_on`, `right_on`으로 각각의 자료에서 컬럼의 키로 쓸 이름을 지정 가능

data1			
	id	col1	col2
0	01	5	1083
1	02	29	1548
2	03	6	1594
3	04	16	1763
4	05	7	1305
5	06	20	1070

data2			
	id	col1	
0	04	4532	
1	05	2599	
2	06	4556	
3	07	3547	

```
# key 다를 경우
data1 = pd.DataFrame({'lkey': ['a', 'b', 'c', 'd'], 'value': [1, 2, 3, 4]}
data2 = pd.DataFrame({'rkey': ['d', 'e', 'a', 'c'], 'value': [5, 6, 7, 8]})

pd.merge(data1, data2, left_on='lkey', right_on='rkey')

lkey  value_x  rkey  value_y
0    a         1    a      7
1    c         3    c      8
2    d         5    d      5
```

데이터 정제 및 준비

- missing data 처리

함수	설명
isnull	누락되거나 NA(not available) 값을 알려주는 불리언 값들이 저장된 객체를 반환
notnull	isnull과 반대되는 메서드
fillna	누락된 데이터에 값을 채우는 메서드. (특정한 값이나 ffill, bfill 같은 보간 메서드 적용)
dropna	누락된 데이터가 있는 축(로우, 컬럼)을 제외시키는 메서드

데이터 정제 및 준비

- `isnull`
 - 누락되거나 NA인 값을 알려주는 불리안 객체 반환
 - `None`, `np.NaN`
- `notnull`
 - 누락되거나 NA인 값이 없다는 것을 알려주는 불리안 객체 반환
 - `None`, `np.NaN`

```
obj = pd.Series([apple, mango, np.nan, None, peach])
```

```
obj.isnull()
```

0	False
1	False
2	True
3	True
4	False

```
dtype: bool
```

```
obj.notnull()
```

0	True
1	True
2	False
3	False
4	True

```
dtype: bool
```

데이터 정제 및 준비

- drop → **dropna(axis=0, how='any', thresh)**
 - 누락되거나 NA인 값을 알려주는 불리안 객체 반환
 - None, np.NaN

frame				
	x1	x2	x3	y
0	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN

frame.dropna()				
	x1	x2	x3	y
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0

# how frame.dropna(how='all')				
	x1	x2	x3	y
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN

데이터 정제 및 준비

- `fillna` → `fillna(value, method='None')`
 - 누락된 데이터를 대신할 값으로 채우거나, 'ffill'이나 'bfill' 같은 보간 메소드를 적용
 - `method`: 보간방법 {'ffill', 'bfill', 'backfill', 'bad', None}

The image shows two DataFrames side-by-side in a Jupyter Notebook interface. The left DataFrame is labeled 'frame' and the right one is labeled 'frame.fillna(0)'. Both DataFrames have columns 'x1', 'x2', 'x3', 'y', and an additional column 'e' which is only present in the second DataFrame.

	x1	x2	x3	y	e
0	NaN	NaN	NaN	NaN	
1	10.0	5.0	40.0	6.0	
2	5.0	2.0	30.0	8.0	
3	20.0	NaN	20.0	6.0	
4	15.0	3.0	10.0	NaN	

	x1	x2	x3	y	e
0	0.0	0.0	0.0	0.0	0.0
1	10.0	5.0	40.0	6.0	0.0
2	5.0	2.0	30.0	8.0	0.0
3	20.0	0.0	20.0	6.0	0.0
4	15.0	3.0	10.0	0.0	0.0

데이터 정제 및 준비

- `fillna` → `fillna(value, method='None')`
 - 컬럼이름을 key로, 대체 값을 value로 하는 dictionary를 전달

frame				
	x1	x2	x3	y
0	NaN	NaN	NaN	NaN
1	10.0	5.0	40.0	6.0
2	5.0	2.0	30.0	8.0
3	20.0	NaN	20.0	6.0
4	15.0	3.0	10.0	NaN

frame.fillna({'x1': 10, 'y': 0})					
	x1	x2	x3	y	e
0	10.0	NaN	NaN	0.0	NaN
1	10.0	5.0	40.0	6.0	NaN
2	5.0	2.0	30.0	8.0	NaN
3	20.0	NaN	20.0	6.0	NaN
4	15.0	3.0	10.0	0.0	NaN

데이터 정제 및 준비

- 중복제거
 - `duplicated()`
 - 각 row가 중복인지(True) 아닌지(False) 알려주는 불리언 series 반환
 - `drop_duplicates()`
 - `duplicated`를 적용한 결과가 False인 것들만 모아서 dataframe 반환

data		
	id	name
0	0001	a
1	0002	b
2	0003	c
3	0001	a

```
data.duplicated()  
0    False  
1    False  
2    False  
3    True  
dtype: bool
```

data. <code>drop_duplicates()</code>		
	id	name
0	0001	a
1	0002	b
2	0003	c

데이터 정제 및 준비

- 중복제거
 - `duplicated()`
 - 각 row가 중복인지(True) 아닌지(False) 알려주는 불리언 series 반환
 - `drop_duplicates()`
 - `duplicated`를 적용한 결과가 False인 것들만 모아서 dataframe 반환

data		
	id	name
0	0001	a
1	0002	b
2	0003	c
3	0001	a

data.drop_duplicates(subset=['id'], keep='last')			
	id	name	phone
1	0002	b	1
2	0003	c	2
3	0001	a	3

기타 메소드

- 데이터 변형
 - replace()
 - get_dummies()
- binning
 - cut(), qcut()
- groupby
 - get_group()
 - agg()
 - filter()

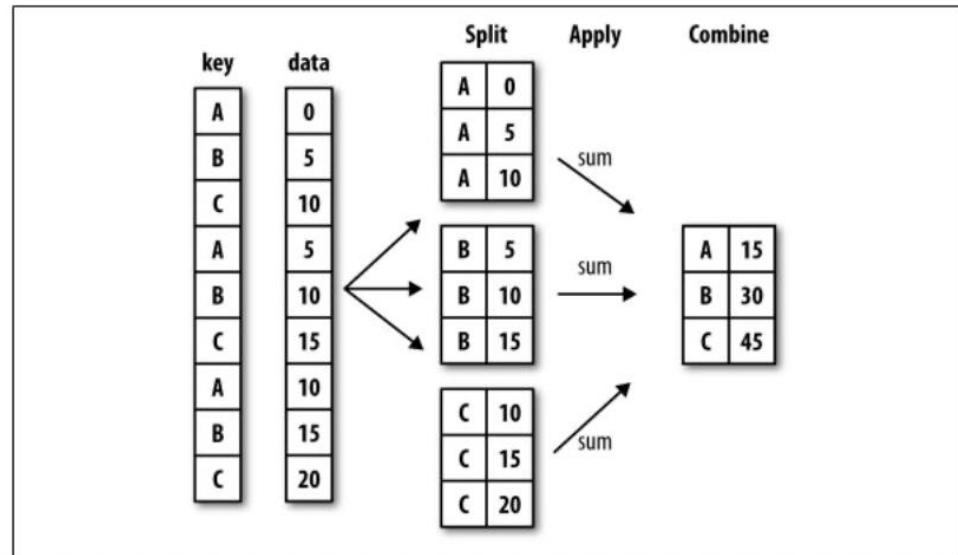
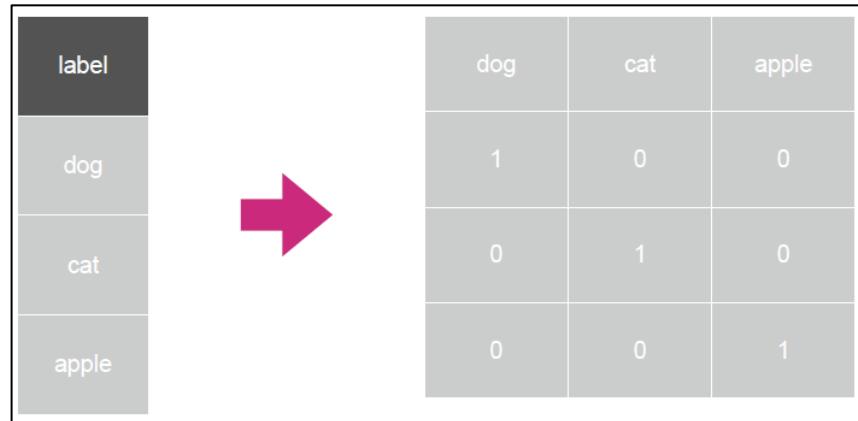


Figure 10-1. Illustration of a group aggregation

Thank you!
Q&A