파이썬 데이터 전처리 모듈 소개
(1) Numpy 모듈

2019. 09. 10.

한양대학교 산업경영공학과

목차

GIGO(Garbage In Garbage Out)

파이썬 데이터 전처리 모듈 소개

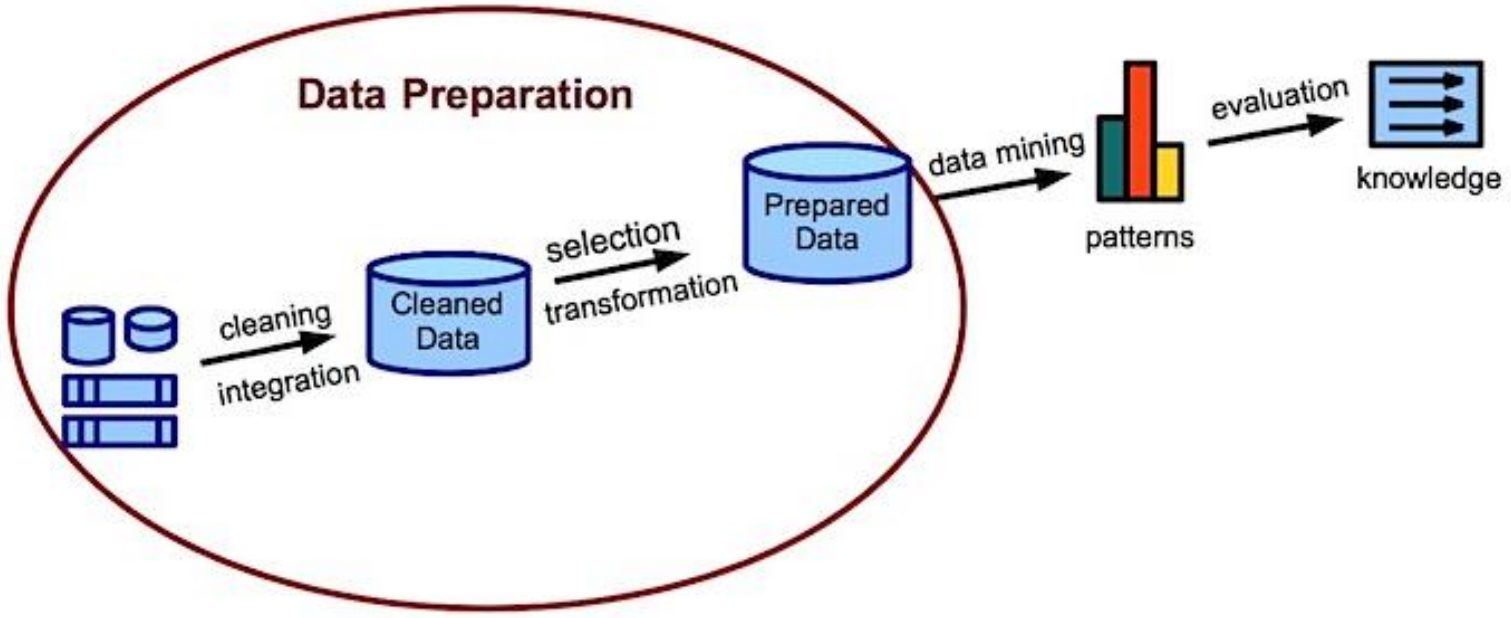
파이썬을 활용한 데이터 전처리에 도움을 주고자 3가지의 주요 모듈에 대한 소개를 하고자 함

- Numpy
- Pandas
- Scikit-Learn

1. 데이터 전처리(data preprocessing)란?

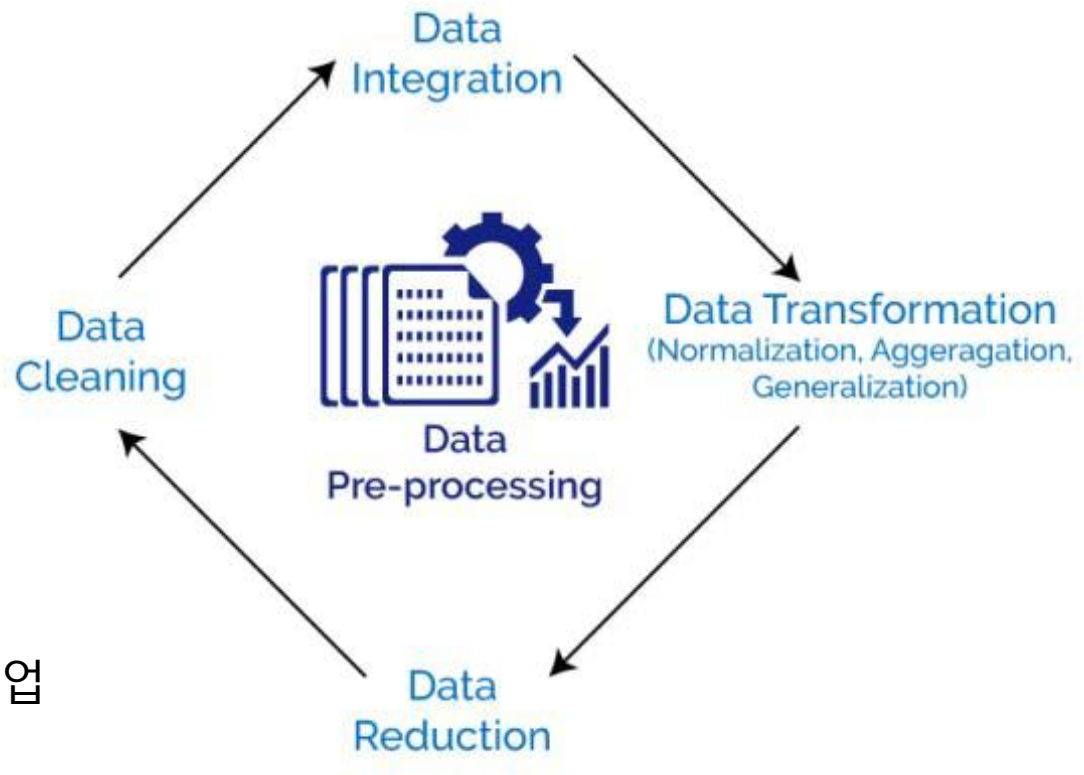
분석 및 처리에 적합한 형식으로 데이터를 조작하는 것을 의미함

데이터 가공(data manipulation), 데이터 핸들링(data handling), 데이터 클리닝(data cleaning), 데이터 준비(data preparation) 등의 표현을 쓰기도 함



1. 데이터 전처리(data preprocessing)란?

- 데이터 정제(data cleaning)
  - 결측치 및 이상치 제거
  - 분석 목적에 맞는 적합한 형식으로의 정제
- 데이터 통합(data integration)
  - 여러 소스로부터 데이터들을 결합하는 문제
- 데이터 변환(data transformation)
  - 데이터의 고유 특성에 맞는 변환
  - 분석 효율을 높이기 위한 작업
- 데이터 정리(data reduction)
  - 방대한 양의 데이터를 의미 있는 수준으로 줄이는 작업
  - PCA,



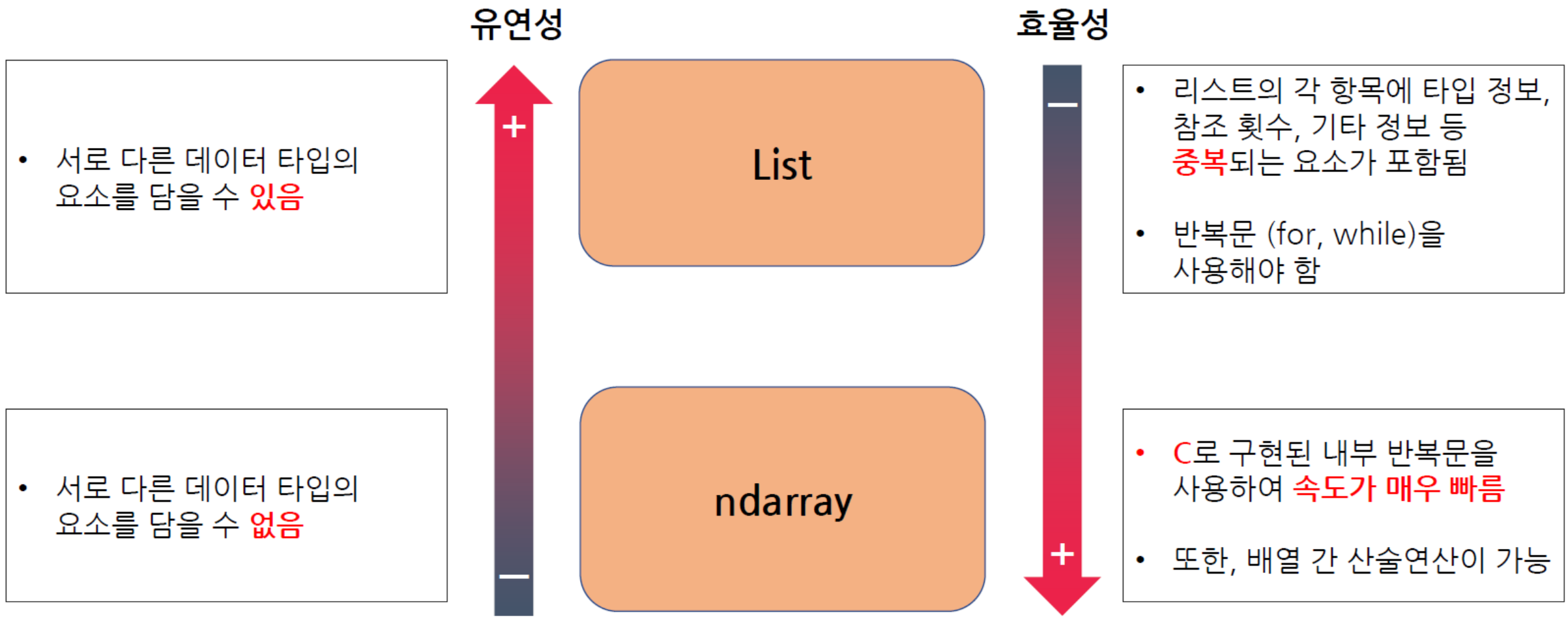
2. Numpy 모듈 소개

Numpy 모듈은 Numerical Python의 약어로, 수치 연산, 배열 및 행렬 연산 처리에 특화된 모듈로 매우 빠른 처리 속도를 보장함

예) 100만개로 구성된 배열 요소의 역수 구하기

-> 반복문이 아닌 numpy 모듈을 활용할 경우 얼마나 빠를까?

3. 배열 기초 - List vs array vs ndarray



3. 배열 기초 - Numpy 표준 데이터 타입

np.array([1,2,3,4], dtype='데이터타입')

자주 사용되는 데이터 타입

- bool: 1바이트로저장된부울값
- intx: $[2^{-x}, 2^{+x}]$사이의 정수(x:8,16,32,64)
- float16: 5비트 지수, 10비트 가수
- float32: 8비트 지수, 23비트 가수
- float64: 11비트 지수, 53비트 가수

요소 크기를 고려하여 데이터 타입을 설정해주는 것이 바람직함

- np.array([530], dtype='int8') #제대로 정의안됨
- np.array([1], dtype='int64') #불필요한 공간 낭비

3. 배열 기초 - Numpy 배열 속성 지정

배열 속성: ndim(차원의 개수), shape(각 차원의 크기), size(전체 배열의 크기), dtype(데이터 타입)

```
# 배열 속성 확인하기

x = np.random.randint(10, size = (3, 4, 5), dtype = 'int8')
x.ndim # 3
x.shape # (3, 4, 5)
x.size # 60 = 3 * 4* 5
x.dtype # int8
```

3. 배열 기초 - Numpy 표준 데이터 타입

규모가 큰 배열의 경우에는 numpy 에 내장된 함수를 사용해 처음부터 배열을 생성하는 것이 더 효율적임

```
# 처음부터 배열 만들기
np.zeros(10, dtype = int) #0으로 채운 배열 생성
np.ones((3, 5), dtype = float) #1로 채운 배열 생성
np.full((3,5), 3.14) # 3.14로 채운 배열 만들기
np.arange(0, 20, 2) # 0에서 시작해 2씩 더해 20까지 채우는 배열 생성
np.linspace(0, 1, 5) # 0과 1 사이에 일정한 간격을 가진 다섯 개의 값으로 채운 배열 만들기
np.random.random((3,3)) # 3*3 크기의 난수 배열 생성
np.random.normal(0, 1, (3, 3)) # 평균 0, 표준 편차 1의 정규 분포를 따르는 3*3 난수 배열
np.random.randint(0, 10, (3, 3)) # [0, 10) 구간의 임의로 정수로 채운 3*3 배열 만들기
np.eye(3) # 크기 3의 단위 행렬 만들기
```

3. 배열 기초 - Numpy 배열 인덱싱

x1 - NumPy array

행 인덱스		0
-6	0	1
-5	1	1
-4	2	5
-3	3	7
-2	4	7
-1	5	5

```
# 배열 인덱싱
x1 = np.random.randint(10, size = 6)
x2 = np.random.randint(10, size = (3, 4))
x3 = np.random.randint(10, size = (3, 4, 5))

x1[0] # 양수 인덱스 (맨 앞 요소)
x1[3] # 세 번째 요소
x1[-1] # 음수 인덱스 (맨 뒤 요소)
x1[-4] # 뒤에서 네 번째 요소

x2[2, 0] # 2행 0열 (리스트 인덱스와 비교)
# list[i][j] / ndarray[i, j]
x3[1, 2, 3] # 1행, 2열, 3번째 요소

x2[0, 0] = 15 # x2의 0행 0열 값을 15로 바꿈
```

3. 배열 기초 - Numpy 배열 슬라이싱

배열 x 의 하위 배열에 접근하기

- x[start: stop: step]
- 값이 입력되지 않으면 start= 0, stop=차원 크기, step=1로 기본 설정됨
- step이 음수이면 역순으로 출력함

```
# 배열 슬라이싱 - 1차원 하위 배열
x = np.arange(10) # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x[:5] # array([0, 1, 2, 3, 4])
x[5:] # array([5, 6, 7, 8, 9])
x[4:7] # array([4, 5, 6])
x[::2] # array([0, 2, 4, 6, 8])
x[1::2] # array([1, 3, 5, 7, 9])
```

```
x[1:8:3] # array([1, 4, 7])
x[::-1] # array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
# 배열 슬라이싱 - 다차원 하위 배열
x = np.random.randint(10, size = (4, 5))
x[:2, :3] # 두 개의 행과 세 개의 열
x[:, ::2] # 모든 행, 한 열 걸러 하나 씩
x[::-1, ::-1] # 행을 역으로, 열도 역으로 (전치 행렬)
```

3. 배열 기초 - Numpy 배열 연결

같은 차원의 배열 연결: concatenate

```
# 같은 차원의 배열 연결
x = np.array([[1,2,3], [4,5,6]])
y = np.array([[3,2,1], [6,5,4]])
np.concatenate([x,y], axis = 0) # 수직 스택
np.concatenate([x,y], axis = 1) # 수평 스택
```

```
In [123]: np.concatenate([x,y], axis = 0) # 행 단위 연결
Out[123]:
array([[1, 2, 3],
       [4, 5, 6],
       [3, 2, 1],
       [6, 5, 4]])

In [124]: np.concatenate([x,y], axis = 1) # 열 단위 연결
Out[124]:
array([[1, 2, 3, 3, 2, 1],
       [4, 5, 6, 6, 5, 4]])
```

혼합된 차원의 배열 연결: vstack(수직 스택) vs hstack(수평 스택)

```
# 혼합된 차원의 배열 연결
x = np.array([[1,2,3], [4,5,6]])
y = np.array([[3,2,1], [6,5,4], [3,2,1]])
z = np.array([[7], [10]])
np.vstack([x, y]) # x와 y의 수직 스택
np.hstack([x, z]) # x와 y의 수평 스택

# np.vstack([x, z]) # 오류 발생 (행 길이가 다름)
# np.hstack([y, z]) # 오류 발생 (열 길이가 다름)
```

```
In [132]: np.vstack([x, y])
Out[132]:
array([[1, 2, 3],
       [4, 5, 6],
       [3, 2, 1],
       [6, 5, 4],
       [3, 2, 1]])

In [133]: np.hstack([x, z])
Out[133]:
array([[ 1,  2,  3,  7],
       [ 4,  5,  6, 10]])
```

3. 배열 기초 - Numpy 배열 분할

vsplit(array, [분할점]): 수직 기준 분할

hsplit(array, [분할점]): 수평 기준 분할

```
x = np.arange(16).reshape((4, 4))
upper, lower = np.vsplit(x, [2]) # 2행을 기준으로 자름
left, right = np.hsplit(x, [2]) # 2열을 기준으로 자름
```

```
In [9]: x
Out[9]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [144]: upper
Out[144]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

In [145]: lower
Out[145]:
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [146]: left
Out[146]:
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
```

```
In [147]: right
Out[147]:
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

루프는 느리다!

```
# 루프는 느리다: 100만개로 구성된 배열 요소의 역수를 구하는 문제
import numpy as np
values = np.random.randint(1, 10, size = 10**6) #100만개 요소를 포함하는 ndarray

import time
t1 = time.time() #time.time(): 현재 시간을 측정
output = np.empty(len(values)) #100만개 요소를 빈 값으로 채움
for i in range(len(values)): #values 들의 인덱스를 반복 (i: values의 index로 해석 가능)
    output[i] = 1.0 / values[i] # i번째 output = 1 / i번째 value
t2 = time.time()
print("루프:", t2 - t1, "초가 걸렸습니다")   루프: 1.8810031414031982 초가 걸렸습니다
```

루프는 느리다!

```
t1 = time.time()
1.0 / values # values에 포함된 모든 요소의 역수
t2 = time.time()
print("유니버설 함수:", t2 - t1, "초가 걸렸습니다")
```

유니버설 함수: 0.0049560070037841 8 초가 걸렸습니다

루프보다 379.54배 빠르다!

4. 비교, 부울 로직

비교 연산자의 결과는 항상 부울(bool) 타입의 배열임

```
# 비교 연산자 - 기초
import numpy as np
x = np.array([1, 2, 3, 4, 5])
x < 3
x != 3
x == 3
M = np.random.randint(1, 10, (3, 4))
M < 6
```

```
In [51]: x < 3
Out[51]: array([ True,  True, False, False, False])

In [52]: x != 3
Out[52]: array([ True,  True, False,  True,  True])

In [53]: x == 3
Out[53]: array([False, False,  True, False, False])

In [54]: M = np.random.randint(1, 10, (3, 4))

In [55]: M < 6
Out[55]:
array([[False,  True,  True, False],
       [ True, False,  True, False],
       [ True, False, False, False]])
```

4. 비교, 부울 로직 - 부울 배열로 작업하기

np.sum(bool_list): bool_list에 포함된 True개수

np.any(bool_list): bool_list에 하나라도 True가 있는지 여부 반환

np.all(bool_list): bool_list에 모든 요소가 True인지 여부 반환

```
# 부울 배열로 작업하기 - 요소 개수 세기
np.sum(M < 6) # 6보다 작은 M 요소 개수
np.sum(M < 6, axis = 0) # 각 열에서 6보다 작은 M 요소 개수
np.sum(M < 6, axis = 1) # 각 행에서 6보다 작은 M 요소 개수
np.sum((M > 5) & (M <= 2)) # Bool & Bool: and 조건 (둘 다 True여야 True)
np.sum((M > 5) | (M <= 2)) # Bool | Bool: or 조건 (둘 중 하나만 True여야 True)

# 부울 배열로 작업하기 - 조건
np.any(M<6) # 6보다 작은 요소가 포함되어 있는가?
np.all(M < 6) # 모든 요소가 6보다 작은가?

# 마스크로서의 부울 배열 - Array[Boolean array]: Boolean array에서 True인 값만 추출
M < 5
M[M<5]
```

5. 인덱싱 및 정렬 - 팬시 인덱싱

팬시 인덱싱은 이미 살펴본 단순 인덱싱과 비슷하지만 단일 스칼라 대신 **인덱스 배열**을 전달함

```
# 팬시 인덱싱
import numpy as np
x = np.random.randint(1, 100, size=10)

[x[3], x[7], x[8]] # 일반적인 인덱싱
ind = [3, 7, 8] # 팬시 인덱싱
x[ind] # x[3, 7, 8]

X = np.arange(12).reshape((3, 4))
row = np.array([0, 1, 2])
col =   np.array([2, 1, 3])
X[row, col] # 0행 2열, 1행 1열, 2행 3열 값
# X[[0, 1, 2], [2, 1, 3]]
```

5. 인덱싱 및 정렬 - 팬시 인덱싱으로 값 변경하기

```
# 팬시 인덱싱으로 값 변경하기
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99 # x[2] = 99, x[1] = 99, x[8] = 99, x[4] = 99
print(x)

x[i] -= 10
print(x)

x = np.zeros(10)
x[[0, 1]] = [4, 6] # x의 0번째, 1번째 값을 [4, 6]으로
print(x)
x[[0, 0]] = [4, 6] # 인덱싱도 순서대로
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```

```
[ 0 89 89  3 89  5  6  7 89  9]
```

```
[4. 6. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
[6. 6. 0. 0. 0. 0. 0. 0. 0. 0.]
```

5. 인덱싱 및 정렬 - Numpy의 빠른 정렬: np.sort

파이썬에서도 리스트를 정렬하는 내장 함수인 sort 와 sorted 가 있지만 numpy 에서 제공하는 sort 함수를 사용하는 것이 좋음

- sort와 np sort는 속도 측면에서는 큰 차이가 없으나, np. sort가 다양한 옵션을 제공

```
# np.sort와 np.argsort
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

```
In [131]: np.sort(x)
Out[131]: array([1, 2, 3, 4, 5])
```

5. 인덱싱 및 정렬 - Numpy의 빠른 정렬: np.sort

Numpy 정렬 알고리즘의 가장 유용한 기능 중 하나는 axis 인수를 사용해 다차원 배열의 특정 행이나 열에 따라 정렬할 수 있다는 것임

```
X = np.random.randint(0, 10, (4, 6))
np.sort(X, axis = 0) # 행 정렬
np.sort(X, axis = 1) # 열 정렬

-np.sort(-X, axis = 0) # 내림차순 행 정렬
```

```
In [140]: X
Out[140]:
array([[4, 7, 6, 3, 2, 2],
       [0, 3, 2, 3, 9, 0],
       [0, 8, 7, 3, 4, 7],
       [1, 5, 3, 6, 7, 1]])

In [141]: np.sort(X, axis = 0)
Out[141]:
array([[0, 3, 2, 3, 2, 0],
       [0, 5, 3, 3, 4, 1],
       [1, 7, 6, 3, 7, 2],
       [4, 8, 7, 6, 9, 7]])

In [142]: np.sort(X, axis = 1)
Out[142]:
array([[2, 2, 3, 4, 6, 7],
       [0, 0, 2, 3, 3, 9],
       [0, 3, 4, 7, 7, 8],
       [1, 1, 3, 5, 6, 7]])
```