1️⃣

week1_1장:파이썬 기반의 머신러닝과 생태계 이해

1장. 파이썬 기반의 머신러닝과 생태계 이해

01. 머신러닝의 개념

: 애플리케이션을 수정하지 않고도 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법을 통칭함.

데이터 기반으로 통계적 신뢰도 강화 & 예측 오류 최소화 위해 다양한 수학적 기법 적용 → 데이터 내의 패턴 스스로 학습/인지 → 신뢰도 있는 예측 결과 도출 → 새로운 예측 모델 이용해 더 정확한 예측 및 의사 결정 도출 & 데이터에 감춰진 새로운 의미와 인사이트 발굴

머신러닝의 분류

지도학습(Supervised Learning)	**비지도학습(Un-supervised Learning)**
분류	클러스터링
회귀	차원 축소
추천 시스템	강화학습
시각/음성 감지/인지	
텍스트 분석, NLP	

데이터 전쟁

머신러닝은 데이터에 매우 의존적이기 때문에 데이터의 중요성이 무엇보다 크다. 가비지 인(Garbage In), 가비지 아웃(Garbage Out), 즉 좋은 품질의 데이터를 갖추지 못한다면 머신러닝의 수행 결과도 좋을 수 없다.

⇒ 최적의 머신러닝 알고리즘과 모델 파라미터 구축하는 능력도 중요하지만, 데이터를 이해하고 효율적으로 가공, 처리, 추출해 최적의 데이터를 기반으로 알고리즘을 구동할 수 있도록 준비하는 능력이 더 중요함.

파이썬과 R 기반의 머신러닝 비교

R: 통계 전용 프로그램 언어 VS. 파이썬: 개발 전문 프로그램 언어

→ 파이썬 추천(딥러닝 프레임워크인 텐서플로, 케라스, 파이토치 등에서 파이썬 우선 정책으로 파이썬 지원하기 때문에, 딥러닝 프레임워크는 파이썬을 중심으로 발전될 가능성이 큼)

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

파이썬 머신러닝을 위한 S/W 설치

1. Anaconda 설치(파이썬으로 작성된 패키지 pip 명령어로 설치 가능하지만, 이 경우 개별 패키지를 별도로 설치해야 하는 불편함이 있음. Anaconda는 파이썬 기반의 머신러닝에 필요한 패키지 일괄적으로 설치할 수 있음)

   → 기본적으로 파이썬 + 넘파이, 판다스, 맷플롯립, 시본, 주피터 노트북까지 함께 설치됨

2. Visual Studio Build Tools

머신러닝 개발을 위해 넘파이와 판다스 이해하는 것 중요함. 머신러닝 애플리케이션을 파이썬으로 개발한다면 대부분의 코드는 사이킷런의 머신러닝 알고리즘에 입력하기 위한 데이터의 추출/가공/변형, 원하는 차원 배열로의 변환을 포함해 머신러닝 알고리즘 처리 결과에 대한 다양한 가공 등으로 구성되는데, **데이터 처리 부분은 대부분 넘파이와 판다스의 몫.**

03. 넘파이(Numerical Python: NumPy)

: 파이썬에서 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 대표적인 패키지

(넘파이는 C/C++과 같은 저수준 언어 기반의 호환 API를 제공하여, 수행 성능이 중요한 부분은 C/C++ 기반의 코드로 작성하고 이를 넘파이에서 호출하는 방식으로 쉽게 통합할 수 있음 - e.g. 구글의 딥러닝 프레임워크 텐서플로)

많은 머신러닝 알고리즘이 넘파이 기반으로 작성돼 있고, 이들 알고리즘의 입력 데이터와 출력 데이터를 넘파이 배열 타입으로 사용한다.

넘파이 ndarray 개요

```
import numpy as np
```

넘파이의 기반 데이터 타입은 ndarray이다. **ndarray**를 이용해 넘파이에서 **다차원 배열**을 쉽게 생성하고 다양한 연산을 수행할 수 있음.

```
array1 = np.array([1,2,3])
print('array1 type:', type(array1))  # array1 type: <class 'numpy.ndarray'>
print('array1 형태:', array1.shape)
# array1 형태: (3,)

array2 = np.array([[1,2,3],
                   [2,3,4]])  ## ([ [], [] ])
print('array2 type:', type(array2))  # array2 type: <class 'numpy.ndarray'>
print('array2 형태:', array2.shape) # array2 형태: (2, 3)
```

```
array3 = np.array([[1,2,3]])
print('array3 type:', type(array3))  # array3 type: <class 'numpy.ndarray'>
print('array3 형태:', array3.shape)  # array3 형태: (1, 3)
```

- array1 = np.array(**[1,2,3]**) → array1 형태: (3,) ⇒ 1차원 array로 3개의 데이터를 가지고 있음. (1차원이니까 행열 개념 x)
- array2 = np.array(**[[1,2,3], [2,3,4]]**) → array2 형태: (2, 3) ⇒ 2차원 array로, 2개의 행(row)와 3개의 열(column)으로 구성 → 총 6개의 데이터
- array3 = np.array(**[[1,2,3]]**) → array3 형태: (1, 3) ⇒ 2차원 array, 1개의 행(row)와 3개의 열(column)으로 구성

```
print('array1: {:0}차원, array2: {:1}차원, array3: {:2}차원'.format(array1.ndim, array2.ndi
m, array3.ndim))
# array1: 1차원, array2: 2차원, array3:  2차원
```

array( ) 함수의 인자로는 파이썬의 리스트 객체가 주로 사용됨. 리스트 [ ]는 1차원이고, 리스트이 리스트[[ ]]는 2차원과 같은 형태로 배열의 차원과 크기를 쉽게 표현할 수 있기 때문이다.

ndarray의 데이터 타입

ndarray 내의 데이터 타입은 같은 데이터 타입만 가능함.

```
list1 = [1,2,3]
print(type(list1))  # <class 'list'>

array1 = np.array(list1)
print(type(array1))  # <class 'numpy.ndarray'>
print(array1, array1.dtype)  # [1 2 3] int64
```

만약 다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변경하면 데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용함.

```
list2 = [1, 2, 'test']
array2 = np.array(list2)
print(array2, array2.dtype)  # ['1' '2' 'test'] <U21

list3= [1, 2, 3.0]
array3= np.array(list3)
print(array3, array3.dtype)  # [1. 2. 3.] float64
```

ndarray 내 데이터값의 타입 변경

```
array_int = np.array([1,2,3])
array_float = array_int.astype('float64')
print(array_float, array_float.dtype)  # [1. 2. 3.] float64

array_int1 = array_float.astype('int32')
print(array_int1, array_int1.dtype)  # [1 2 3] int32

array_float1 = np.array([1.1, 2.1, 3.1])
array_int2 = array_float1.astype('int32')
print(array_int2, array_int2.dtype)  # [1 2 3] int32
```

ndarray를 편리하게 생성하기 - arange, zeros, ones

`arange()` array를 range( )로 표현하는 것

```
sequence_array = np.arange(10)
print(sequence_array)  # [0 1 2 3 4 5 6 7 8 9]
print(sequence_array.dtype, sequence_array.shape)  # int64 (10,)
```

`zeros()` 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray로 반환

`ones()` 유사하게 모든 값을 1로 채운 ndarray 반환

```
zero_array = np.zeros((3,2), dtype='int32')
print(zero_array)
'''
[[0 0]
 [0 0]
 [0 0]]
'''
print(zero_array.dtype, zero_array.shape)  # int32 (3, 2)

one_array = np.ones((3,2))
print(one_array)
'''
[[1. 1.]
 [1. 1.]
 [1. 1.]]
'''
print(one_array.dtype, one_array.shape)  # float64 (3, 2)
```

ndarray의 차원과 크기를 변경하는 reshape

`reshape()` ndarray를 특정 차원 및 크기로 변환함, 변환을 원하는 크기를 함수 인자로 부여하면 됨

```
array1 = np.arange(10)
print('array1:\n', array1)
'''
array1:
 [0 1 2 3 4 5 6 7 8 9]
'''

array2 = array1.reshape(2,5)
print('array2:\n', array2)
'''
array2:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
'''

array3 = array1.reshape(5,2)
print('array3:\n', array3)
'''
array3:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
'''
```

인자를 -1로 사용하면 원래 ndarray와 호환되는 새로운 shape로 변환해 줌.

```
array1 = np.arange(10)
print(array1)  # [0 1 2 3 4 5 6 7 8 9]

array2 = array1.reshape(-1,5)  # 고정된 5개의 칼럼에 맞는 로우를 자동으로 새롭게 생성해 변환하라는 의미
print('array2 shape:', array2.shape)  # array2 shape: (2, 5)

array3 = array1.reshape(5,-1)
print('array3 shape:', array3.shape)  # array3 shape: (5, 2)
```

-1 인자는 `reshape(-1, 1)`와 같은 형태로 자주 사용됨. `reshape(-1, 1)` 은 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 로우를 가지되 반드시 1개의 칼럼을 가진 ndarray로 변환됨을 보장함.

여러 개의 넘파이 ndarray는 stack이나 concat으로 결합할 때 각각의 ndarray의 형태를 통일해 유용하게 사용됨. 아래 예제는 3차원을 2차원으로, 1차원을 2차원으로 변경한 예시.

(ndarray는 tolist() 메서드를 이용해 리스트 자료형으로 변환할 수 있음. 때로는 리스트 자료형을 print를 이용해 출력할 때 시각적으로 더 이해하기 쉬울 수 있어 ndarray를 리스트로 변환해 출력함)

```
array1 = np.arange(8)
array3d = array1.reshape((2, 2, 2))
print(array3d)
'''
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
'''
print('array3d:\n', array3d.tolist())
'''
array3d:
 [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
'''

# 3차원 ndarray를 2차원 ndarray로 변환
array5 = array3d.reshape(-1,1)
print('array5:\n', array5.tolist())
print('array5.shape:', array5.shape)  # array5.shape: (8, 1)
'''
array5:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
'''

# 1차원 ndarray를 2차원 ndarray로 변환
array6 = array1.reshape(-1,1)
print('array6:\n', array6.tolist())
print('array6.shape:', array6.shape)  # array6.shape: (8, 1)
'''
array6:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
'''
```

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

1. 특정 데이터만 추출: 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터가 반환됨
2. 슬라이싱: 슬라이싱은 연속된 인덱스상의 ndarray를 추출하는 방식. ‘:’ 기호 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의

ndarray를 반환함.

3. 팬시 인덱스(Fancy Indexing): 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray를 반환함.
4. 불린 인덱싱: 특정 조건에 해당하는지 여부인 True/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환함.

단일 값 추출

한 개의 데이터 추출 방법 → narray 객체에 해당하는 위치의 인덱스 값을 [ ] 안에 입력하면 됨.

```
# 1~9까지 1차원의 ndarray 생성
array1 = np.arange(start=1, stop=10)
print('array1:', array1)  # array1: [1 2 3 4 5 6 7 8 9]

# index는 0부터 시작하므로 array1[2]는 3번째 index 위치의 데이터값을 의미
value = array1[2]
print('value:', value)  # value: 3
print(type(value))  # <class 'numpy.int64'>
# array1[2]는 더이상 ndarray 타입이 아닌, ndarray 내의 데이터값을 의미함
```

인덱스에 마이너스 기호를 사용하면 맨 뒤에서부터 데이터를 추출할 수 있음. 인덱스 -1은 맨 뒤의 데이터값 의미함.

```
print('맨 뒤의 값:', array1[-1], '맨 뒤에서 두번재 값:', array1[-2])
# 맨 뒤의 값: 9 맨 뒤에서 두번재 값: 8
```

단일 인덱스를 이용해 ndarray 내의 데이터 값 **수정** 가능함

```
array1[0] = 9
array1[8] = 0
print('array1:', array1)  # array1: [9 2 3 4 5 6 7 8 0]
```

다차원 ndarray 내에서 단일 값 추출

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3, 3)  # 1차원의 ndarray를 2차원의 3*3 ndarray로 변환
print(array2d)
'''[[1 2 3]
   [4 5 6]
   [7 8 9]]
```

```
'''
print('(row=0,col=0) index 가리키는 값:', array2d[0,0])
print('(row=0,col=1) index 가리키는 값:', array2d[0,1] )
print('(row=1,col=0) index 가리키는 값:', array2d[1,0] )
print('(row=2,col=2) index 가리키는 값:', array2d[2,2] )
'''
(row=0,col=0) index 가리키는 값: 1
(row=0,col=1) index 가리키는 값: 2
(row=1,col=0) index 가리키는 값: 4
(row=2,col=2) index 가리키는 값: 9
'''
```

axis 0은 로우 방향의 축 의미, **axis1은 칼럼** 방향의 축 의미함 → 2차원이므로 axis 0, axis 1로 구분되며 3차원 ndarray는 axis 0, axis 1, axis 2(행, 열, 높이) 3개의 축을 가지게 됨.

슬라이싱

':' 기호를 이용해 연속한 데이터를 슬라이싱해서 추출할 수 있음.

단일값 추출을 제외하고는 ndarray 타입으로 추출 됨.

```
array1 = np.arange(start=1, stop=10)
array3 = array1[0:3]
print(array3)  # [1 2 3]
print(type(array3))  # <class 'numpy.ndarray'>
```

- : 앞에 시작 인덱스 생략 → 자동으로 맨 처음 인덱스 0으로 간주
- : 기호 뒤에 종료 인덱스 생략 → 자동으로 맨 마지막 인덱스로 간주
- : 기호 앞/뒤에 시작/종료 인덱스 생략 → 자동으로 맨 처음/맨 마지막 인덱스로 간주

```
array1 = np.arange(start=1, stop=10)
array4 = array1[:3]
print(array4)  # [1 2 3]

array5 = array1[3:]
print(array5)  # [4 5 6 7 8 9]

array6 = array1[:]
print(array6)  # [1 2 3 4 5 6 7 8 9]
```

다차원 슬라이싱 → 콤마(,)로 로우와 칼럼 인덱스 지칭하는 부분 다름

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3,3)
print('array2d:\n', array2d)
'''
array2d:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
'''

print('array2d[0:2, 0:2] \n', array2d[0:2, 0:2])
print('array2d[1:3, 0:3] \n', array2d[1:3, 0:3])
print('array2d[1:3, :] \n', array2d[1:3, :])
print('array2d[:, :] \n', array2d[:, :])
print('array2d[:2, 1:] \n', array2d[:2, 1:])
print('array2d[:2, 0] \n', array2d[:2, 0])
'''
array2d[0:2, 0:2]
 [[1 2]
 [4 5]]
array2d[1:3, 0:3]
 [[4 5 6]
 [7 8 9]]
array2d[1:3, :]
 [[4 5 6]
 [7 8 9]]
array2d[:, :]
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
array2d[:2, 1:]
 [[2 3]
 [5 6]]
array2d[:2, 0]
 [1 4]
'''

# 2차원 뒤에 오는 인덱스 없애면 1차원 ndarray 반환
print(array2d[0])  # [1 2 3]
print(array2d[1])  # [4 5 6]
print('array2d[0] shape:', array2d[0].shape, 'array2d[1] shape:', array2d[1].shape)
# array2d[0] shape: (3,) array2d[1] shape: (3,)
```

팬시 인덱싱: 리스트나 ndarray로 인덱스 집합 지정 → 해당 ndarray 반환

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3,3)

array3 = array2d[[0,1],2]
```

```
print('array2d[[0,1], 2] => ',array3.tolist())  # array2d[[0,1], 2] =>  [3, 6]

array4 = array2d[[0,1], 0:2]
print('array2d[[0,1], 0:2] => ',array4.tolist())  # array2d[[0,1], 0:2] =>  [[1, 2], [4,
 5]]

array5 = array2d[[0,1]]
print('array2d[[0,1]] => ',array5.tolist())  # array2d[[0,1]] =>  [[1, 2, 3], [4, 5, 6]]
# ( (0,0), (0,1) ), ( (1,0),(1,1) ) 인덱싱이 적용됨
```

불린 인덱싱: 조건 필터링과 검색을 동시에 할 수 있어 매우 자주 사용되는 인덱싱 방식, ndarray의 인덱스를 지정하는 [ ] 내에 조건문을 그대로 기재하면 됨.

```
array1d = np.arange(start=1, stop=10)
# []안에 array1d > 5 Boolean indexing 적용
array3 = array1d[array1d>5]
print('array1d > 5 불린 인덱싱 결과 값 :', array3)
# array1d > 5 불린 인덱싱 결과 값 : [6 7 8 9]
```

```
array1d > 5
# array([False, False, False, False, False,  True,  True,  True,  True])
```

→ True 값이 있는 위치 인덱스 값으로 자동 변환해 해당하는 인덱스 위치의 데이터 반환

```
boolean_indexes = np.array([False, False, False, False, False,  True,  True,  True,  Tru
e])
array3 = array1d[boolean_indexes]
print('불린 인덱스로 필터링 결과 :', array3)
# 불린 인덱스로 필터링 결과 : [6 7 8 9]
```

```
# 직접 인덱스 집합 만들어 대입

indexes = np.array([5,6,7,8])
array4 = array1d[ indexes ]
print('일반 인덱스로 필터링 결과 :',array4)
# 일반 인덱스로 필터링 결과 : [6 7 8 9]
```

행렬의 정렬 - sort()와 argsort()

행렬 정렬

`np.sort()` 원 행렬 그대로 유지한 채 원 행렬의 정렬된 행렬 반환

`ndarray.sort()` 원 행렬 자체를 정렬한 형태로 변환, 반환 값은 None이 됨(행렬 자체를 정렬한 값으로 변환)

```
org_array = np.array([ 3, 1, 9, 5])
print('원본 행렬:', org_array)  # 원본 행렬: [3 1 9 5]

# np.sort()로 정렬
sort_array1 = np.sort(org_array)
print ('np.sort( ) 호출 후 반환된 정렬 행렬:', sort_array1)
print('np.sort( ) 호출 후 원본 행렬:', org_array)
'''
np.sort( ) 호출 후 반환된 정렬 행렬: [1 3 5 9]
np.sort( ) 호출 후 원본 행렬: [3 1 9 5]
'''
# ndarray.sort()로 정렬
sort_array2 = org_array.sort()
print('org_array.sort( ) 호출 후 반환된 행렬:', sort_array2)
print('org_array.sort( ) 호출 후 원본 행렬:', org_array)
'''
org_array.sort( ) 호출 후 반환된 행렬: None
org_array.sort( ) 호출 후 원본 행렬: [1 3 5 9]
'''
```

내림차순으로 정렬하기 위해서는 [::-1] 적용

```
sort_array1_desc = np.sort(org_array)[::-1]
print ('내림차순으로 정렬:', sort_array1_desc)
# 내림차순으로 정렬: [9 5 3 1]
```

행렬이 2차원 이상일 경우 axis 축 값 설정을 통해 로우 방향, 또는 칼럼 방향으로 정렬을 수행할 수 있음.

```
array2d = np.array([[8, 12],
                    [7, 1 ]])

sort_array2d_axis0 = np.sort(array2d, axis=0)
print('로우 방향으로 정렬:\n', sort_array2d_axis0)

sort_array2d_axis1 = np.sort(array2d, axis=1)
print('컬럼 방향으로 정렬:\n', sort_array2d_axis1)
'''
로우 방향으로 정렬:
 [[ 7  1]
```

```
 [ 8 12]]
컬럼 방향으로 정렬:
 [[ 8 12]
 [ 1  7]]
'''
```

정렬된 행렬의 인덱스 반환하기

: 원본 행렬이 정렬되었을 때 기존 원본 행렬의 원소에 대한 인덱스 필요할 때 → `np.argsort()`: 정렬 행렬의 원본 행렬 인덱스를 ndarray 형태롤 반환

```
org_array = np.array([ 3, 1, 9, 5])
sort_indices = np.argsort(org_array)
print(type(sort_indices))  # <class 'numpy.ndarray'>
print('행렬 정렬 시 원본 행렬의 인덱스:', sort_indices)  # 행렬 정렬 시 원본 행렬의 인덱스: [1 0 3 2]
```

```
# 내림차순 정렬 -> [::-1]
org_array = np.array([ 3, 1, 9, 5])
sort_indices_desc = np.argsort(org_array)[::-1]
print(type(sort_indices))  # <class 'numpy.ndarray'>
print('행렬 내림차순 정렬 시 원본 행렬의 인덱스:', sort_indices_desc)
# 행렬 내림차순 정렬 시 원본 행렬의 인덱스: [2 3 0 1]
```

선형대수 연산 - 행렬 내적과 전치 행렬 구하기

행렬 내적(행렬 곱)

행렬 내적은 행렬 곱으로, 두 행렬 A와 B의 내적은 `np.dot()` 을 이용해 계산할 수 있음. 내적은 왼쪽 행렬의 로우(행)와 오른쪽 행렬의 칼럼의 원소들을 순차적으로 곱한 뒤 그 결과들을 모두 더한 값이다.

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
B = np.array([[7, 8],
              [9, 10],
              [11, 12]])
dot_product = np.dot(A, B)
print('행렬 내적 결과:\n', dot_product)
'''
행렬 내적 결과:
 [[ 58  64]
 [139 154]]
'''
```

전치 행렬

: 원 행렬에서 행과 열 위치를 교환한 원소로 구성한 행렬을 그 행렬의 전치 행렬이라고 함, `transpose()` 이용.

(2*2 행렬 A가 있다하면, A 행렬의 1행 2열의 원소를 2행 1열의 원소로, 2행 1열의 원소는 1행 2열의 원소로 교환하는 것) → A^T 로 표현함

```
A = np.array([[1, 2],
              [3, 4]])
transpose_mat = np.transpose(A)
print('A의 전치 행렬:\n', transpose_mat)
'''
A의 전치 행렬:
 [[1 3]
 [2 4]]
'''
```

numpy 정리

`type(array1)`	데이터 타입
`array1.shape`	ndarray의 크기 ⇒ (행, 열) ⇒ 차원까지 알 수 있음
`np.array()`	ndarray 변환을 원하는 객체를 인자로 입력하면 ndarray 반환
`ndarray.ndim`	차원 수 확인
`array1.dtype`	ndarray 내의 데이터 타입 확인
`array.astype('float64')`	ndarray 내 데이터값의 타입 변경, ( ) 안에 원하는 타입을 문자열로 지정
`arange()`	0부터 (입력한 값-1)까지의 연속 숫자 값으로 구성된 1차원 ndarray 만들어줌 → 인자로 stop 값 부여
`zeros()`	튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray로 반환
`ones()`	모든 값을 1로 채운 해당 shape를 가진ndarray 반환
`reshape()`	ndarray를 특정 차원 및 크기로 변환함, 변환을 원하는 크기를 함수 인자로 부여하면 됨 인자를 -1로 사용하면 원래 ndarray와 호환되는 새로운 shape로 변환해 줌.
단일 인덱스	`array1[2]`
단일 인덱스로 값 수정	`array1[0] = 9`
다차원 ndarray 내 단일	`array2d[0,0]`

값	
다차원 슬라이싱	`array2d[0:2, 0:2]`
팬시 인덱싱	`array2[[0,1], 0:2]` 리스트나 ndarray로 인덱스 집합 지정
불린 인덱싱	`array1d[array1d>5]` [] 안에 조건 넣어줌
`np.sort()`	원 행렬 그대로 유지한 채 원 행렬의 정렬된 행렬 반환
`ndarray.sort()`	원 행렬 자체를 정렬한 형태로 변환, 반환 값은 None
`np.argsort()`	정렬 행렬의 원본 행렬 인덱스를 ndarray 형태롤 반환
`np.dot()`	행렬 내적(행렬 곱)
`transpose()`	전치 행렬

04. 데이터 핸들링 - 판다스

일반적으로 데이터 셋은 행과 열로 이뤄진 2차원 데이터이다. 판다스는 파이썬의 리스트, 컬렉션, 넘파이 등의 내부 데이터 뿐만 아니라 CSV 등의 파일을 쉽게 DataFrame으로 변경해 데이터의 가공/분석을 편리하게 해줌.

판다스의 핵심 객체는 DataFrame(행과 열로 이뤄진 2차원 데이터를 담는 데이터 구조체)이다. Index는 개별 데이터를 고유하게 식별하는 Key 값, Series는 칼럼이 하나, DataFrame은 칼럼이 여러 개인 데이터 구조체이다.

판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

```
import pandas as pd
```

판다스는 다양한 포맷으로 된 파일을 DataFrame으로 로딩할 수 있는 편리한 API 제공함(e.g. `read_csv()`)

```
titanic_df = pd.read_csv('titanic_train.csv')
titanic_df.head(3)
```

```
print('titanic 변수 type:',type(titanic_df))  # titanic 변수 type: <class 'pandas.core.fram
e.DataFrame'>
```

```
print('DataFrame 크기:', titanic_df.shape)  # DataFrame 크기: (891, 12)
# 891개의 로우와 12개의 칼럼으로 구성됨
```

```
titanic_df.info()
'''
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
'''
```

- RangeIndex: DataFrame의 index 범위를 나타냄 → 전체 row 수 알 수 있음
- Dtype(int64, object, float64...): 칼럼별 데이터 타입

```
value_counts = titanic_df['Pclass'].value_counts()
print(value_counts)
'''
3    491
1    216
2    184
Name: Pclass, dtype: int64
'''
```

```
titanic_pclass = titanic_df['Pclass']
print(type(titanic_pclass))  # <class 'pandas.core.series.Series'>
```

↑ df의 [ ] 연산자 내부에 칼럼명을 입력하면 해당 칼럼에 해당하는 series 객체를 반환함. series는 index와 단 하나의 칼럼으로 구성된 데이터 세트이다.

```
titanic_pclass.head()
'''
0    3
1    1
2    3
3    1
4    3
Name: Pclass, dtype: int64
'''
```

→ 첫번째 열: series의 index, 두번재 열: series의 데이터 값(해당 칼럼의 데이터값)

++ 인덱스는 df, series가 만들어진 후에도 변경할 수 있음. 또한 숫자형뿐 아니라 문자열도 가능함. 단, 모든 인덱스는 고유성이 보장돼야 함.

DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

DataFrame과 넘파이 ndarray 상호 간의 변환은 매우 빈번하게 발생함.

넘파이 ndarray, 리스트, 딕셔너리를 DataFrame으로 변환하기

DataFrame은 칼럼명을 가지고 있음. 그래서 DataFrame으로 변환 시 칼럼명을 지정해줘야 함.

```
import numpy as np

col_name1 = ['col1']
list1 = [1,2,3]
array1 = np.array(list1)
print('array1 shape:',array1.shape)  # array1 shape: (3,)

# 리스트를 이용해 dataframe 생성
df_list1 = pd.DataFrame(list1, columns=col_name1)
print('1차원 리스트로 만든 DataFrame:\n', df_list1)
'''
1차원 리스트로 만든 DataFrame:
    col1
0     1
1     2
2     3
'''

# 넘파이 ndarray를 이용해 DataFrame 생성
df_array1 = pd.DataFrame(array1, columns=col_name1)
```

```
print('1차원 ndarray로 만든 DataFrame:\n', df_array1)
'''
1차원 ndarray로 만든 DataFrame:
    col1
0     1
1     2
2     3
'''
```

2차원 형태 데이터 기반으로 DF 생성(2행 3열 리스트와 ndarray) → 3개의 컬럼명 필요

```
# 3개의 컬럼명이 필요함.
col_name2=['col1', 'col2', 'col3']

# 2행x3열 형태의 리스트와 ndarray 생성 한 뒤 이를 DataFrame으로 변환.
list2 = [[1, 2, 3],
         [11, 12, 13]]
array2 = np.array(list2)
print('array2 shape:', array2.shape )

df_list2 = pd.DataFrame(list2, columns=col_name2)
print('2차원 리스트로 만든 DataFrame:\n', df_list2)
df_array2 = pd.DataFrame(array2, columns=col_name2)
print('2차원 ndarray로 만든 DataFrame:\n', df_array2)
'''
array2 shape: (2, 3)
2차원 리스트로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    12    13
2차원 ndarray로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    12    13
'''
```

딕셔너리를 df로 변환(키→칼럼명, 값(value)→키에 해당하는 칼럼 데이터로 변환)

```
# Key는 컬럼명으로 매핑, Value는 리스트 형(또는 ndarray) 칼럼 데이터로 매핑
dict = {'col1':[1, 11], 'col2':[2, 22], 'col3':[3, 33]}
df_dict = pd.DataFrame(dict)
print('딕셔너리로 만든 DataFrame:\n', df_dict)
'''
딕셔너리로 만든 DataFrame:
    col1  col2  col3
0     1     2     3
1    11    22    33
'''
```

DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

values를 이용한 ndarray로의 변환은 매우 많이 사용되니까 기억해두기!

```
# DataFrame을 ndarray로 변환 -> values 사용
array3 = df_dict.values
print(array3)
'''
[[ 1  2  3]
 [11 22 33]]
'''
print('df_dict.values 타입:', type(array3), 'df_dict.values shape:', array3.shape)
# df_dict.values 타입: <class 'numpy.ndarray'> df_dict.values shape: (2, 3)
```

```
# DataFrame을 리스트로 변환 -> tolist() 사용
list3 = df_dict.values.tolist()
print(list3)  # [[1, 2, 3], [11, 22, 33]]
print('df_dict.values.tolist() 타입:', type(list3))  # df_dict.values.tolist() 타입: <class
'list'>

# DataFrame을 딕셔너리로 변환 -> to_dict() 사용
dict3 = df_dict.to_dict('list')
print('\n df_dict.to_dict() 타입:', type(dict3))  # df_dict.to_dict() 타입: <class 'dict'>
print(dict3)  # {'col1': [1, 11], 'col2': [2, 22], 'col3': [3, 33]}
```

DataFrame의 칼럼 데이터 세트 생성과 수정

```
# titanic에 새로운 칼럼 Age_0 추가, 일괄적으로 0 값 할당
titanic_df['Age_0'] = 0
```

→ 새로운 칼럼명 ‘Age_0’으로 모든 데이터값이 0으로 할당된 series가 기존 df에 추가됨

```
# 기존 칼럼의 series 데이터 이용해 새로운 칼럼 series 만들기
titanic_df['Age_by_10'] = titanic_df['Age']*10
titanic_df['Family_No'] = titanic_df['SibSp'] + titanic_df['Parch']+1
```

```
# df 내의 기존 칼럼 값 일괄 업데이트 (기존값 + 100으로 업데이트)
titanic_df['Age_by_10'] = titanic_df['Age_by_10'] + 100
```

DataFrame 데이터 삭제

`drop()`

- axis = 특정 로우 또는 칼럼 드롭(axis 0은 로우(인덱스) 방향, axis 1은 칼럼 방향 축)
- inplace = False(디폴트 값, 원본 df 유지하고 드롭된 df 새로 반환)
- 여러개 삭제 → 리스트 형태로 삭제하고자 하는 칼럼명 입력

```
titanic_drop_df = titanic_df.drop('Age_0', axis=1 )
titanic_drop_df.head(3)  # 'Age_0' 칼럼 없는 df 반환됨
```

```
drop_result = titanic_df.drop(['Age_0', 'Age_by_10', 'Family_No'], axis=1, inplace=True)
print(' inplace=True 로 drop 후 반환된 값:',drop_result)
# inplace=True 로 drop 후 반환된 값: None -> 자기 자신을 반환 객체로 설정하면 안됨.
```

```
# 행(로우) 삭제
titanic_df.drop([0,1,2], axis=0, inplace=True)
```

Index 객체

: df, series의 레코드를 고유하게 식별하는 객체

```
# 원본 파일 다시 로딩
titanic_df = pd.read_csv('titanic_train.csv')
# Index 객체 추출
indexes = titanic_df.index
print(indexes)  # RangeIndex(start=0, stop=891, step=1)
# Index 객체를 실제 값 array로 변환
print('Index 객체 array값:\n',indexes.values)
'''
Index 객체 array값:
 [  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  30  31  32  33  34  35
 ...
 882 883 884 885 886 887 888 889 890]
'''
```

ndarray와 유사하게 단일 값 변환 및 슬라이싱도 가능함.

```
print(type(indexes.values))  # <class 'numpy.ndarray'>
print(indexes.values.shape)  # (891,)
print(indexes[:5].values)  # [0 1 2 3 4]
print(indexes.values[:5])  # [0 1 2 3 4]
print(indexes[6])  # 6
```

하지만 한 번 만들어진 df 및 series의 인덱스 변경x → index 객체의 값 변경하는 작업 x

series 객체에 연산 함수 적용할 때 index는 연산에서 제외됨, 인덱스는 오직 식별용으로만 사용됨

```
series_fair = titanic_df['Fare']
print('Fair Series max 값:', series_fair.max())  # Fair Series max 값: 512.3292
print('Fair Series sum 값:', series_fair.sum())  # Fair Series sum 값: 28693.9493
print('sum() Fair Series:', sum(series_fair))  # sum() Fair Series: 28693.949299999967
print('Fair Series + 3:\n',(series_fair + 3).head(3) )
'''
Fair Series + 3:
0    10.2500
1    74.2833
2    10.9250
Name: Fare, dtype: float64
'''
```

df 및 series에 reset_index() 메서드 수행 → 새롭게 인덱스를 연속 숫자 형으로 할당 & 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가 함 (주로 인덱스가 연속된 int 숫자형 데이터가 아닐 때 사용)

```
titanic_reset_df = titanic_df.reset_index(inplace=False)
titanic_reset_df.head()
```

데이터 셀렉션 및 필터링

- 넘파이: [ ] 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱을 통해 데이터 추출
- 판다스: ~~ix[ ]~~, **iloc[ ], loc[ ]**연산자 통해 동일 작업 수행

DataFrame의 [ ] 연산자

DF 바로 뒤의 [ ] → 칼럼 명 문자, (또는 인덱스로 변환 가능한 표현식) ⇒ **칼럼 지정 연산자**

```
print('단일 컬럼 데이터 추출:\n', titanic_df[ 'Pclass' ].head(3))
'''
단일 컬럼 데이터 추출:
0    3
1    1
2    3
Name: Pclass, dtype: int64
'''
print('\n여러 컬럼들의 데이터 추출:\n', titanic_df[ ['Survived', 'Pclass'] ].head(3))
'''
여러 컬럼들의 데이터 추출:
    Survived  Pclass
0         0       3
1         1       1
2         1       3
'''
print('[ ] 안에 숫자 index는 KeyError 오류 발생:\n', titanic_df[0]) -> ERROR
```

판다스의 인덱스 형태로 변환 가능한 표현식으로 [ ] 내에 입력 가능함.

```
# df의 처음 2개의 데이터 추출 - 슬라이싱
titanic_df[0:2]
# 불린 인덱싱 표현
titanic_df[titanic_df['Pclass']==3].head()
```

👉 - DF 바로 뒤의 [ ] 연산자는 넘파이의 [ ]나 Series의 [ ]와 다름
- DF 바로 뒤의 [ ] 내 입력 값은 **칼럼명**(→ 칼럼 지정 연산에 사용) 혹은 **불린 인덱스** 용도로만 사용
- DF[0:2] 같은 슬라이싱 연산으로 데이터 추출하는 방식은 사용하지 않는게 좋음

~~DataFrame의 ix[ ] 연산자~~

명칭 기반 인덱싱과 위치 기반 인덱싱의 구분

- 명칭 기반 → 칼럼의 명칭을 기반으로 위치를 지정하는 방식 ⇒ loc[ ]
- 위치 기반 → 0을 출발점으로 행과 열의 위치를 기반으로 데이터 지정하는 방식 ⇒ iloc[ ]
- 불린 인덱싱은 조건을 기술하므로 이에 제약 받지 않음

DataFrame iloc[ ] 연산자

위치 기반 인덱싱 → 행과 열 위치 값으로 정수형 값을 지정해 원하는 데이터 반환

```
data = {'Name': ['Chulmin', 'Eunkyung','Jinwoong','Soobeom'],
        'Year': [2011, 2016, 2015, 2015],
        'Gender': ['Male', 'Female', 'Male', 'Male']
       }
data_df = pd.DataFrame(data, index=['one','two','three','four'])
data_df
'''
Name  Year  Gender
one Chulmin 2011  Male
two Eunkyung  2016  Female
three Jinwoong  2015  Male
four  Soobeom 2015  Male
'''
```

```
data_df.iloc[0,0]  # Chulmin
```

DataFrame loc[ ] 연산자

명칭 기반 인덱싱 → (index 값, 칼럼 명)

- index가 숫자 형일 수 있기 때문에 무조건 문자열 입력해야한다는 선입견 가져서는 안됨

```
data_df.loc['one','Name']  # Chulmin
```

```
print('위치기반 iloc slicing\n', data_df.iloc[0:1, 0],'\n')
'''
위치기반 iloc slicing
 one    Chulmin
Name: Name, dtype: object
'''
print('명칭기반 loc slicing\n', data_df.loc['one':'two', 'Name'])
'''
명칭기반 loc slicing
 one     Chulmin
two    Eunkyung
Name: Name, dtype: object
'''
-> 명칭 기반은 위치 기반과 다르게 종료 값 포함한 범위 반환
```

불린 인덱싱

iloc[ ]는 정수형 값이 아닌 불린 값에 대해서는 지원x → 불린 인덱싱x

```
titanic_df = pd.read_csv('titanic_train.csv')
titanic_boolean = titanic_df[titanic_df['Age'] > 60 ]
print(type(titanic_boolean))  # <class 'pandas.core.frame.DataFrame'>
titanic_boolean  # age가 60 보다 큰 데이터 모두 반환
```

```
# 60세 이상 승객의 나이와 이름만 추출

titanic_df[titanic_df['Age']>60][['Name','Age']].head()  # 괄호 안에 다시 괄호 -> 리스트로 칼럼
 묶어줌

# loc로 추출
titanic_df.loc[titanic_df['Age']>60, ['Name','Age']].head()  # name, age는 칼럼 위치에
```

복합 조건 → 개별 조건 ( )로 묶고, 복합 조건 연산자 사용

- and 조건일 때는 &
- or 조건일 때는 |
- Not 조건일 때는 ~

```
# 60세 이상, 선실 1등급, 여성
titanic_df[ (titanic_df['Age'] > 60) & (titanic_df['Sex']=='female') & (titanic_df['Pclas
s']== 1) ]

# 개별 조건 변수에 할당 -> 변수들 결합해서 불린 인덱싱 수행
cond1 = titanic_df['Age'] > 60
cond2 = titanic_df['Pclass']==1
cond3 = titanic_df['Sex']=='female'
titanic_df[ cond1 & cond2 & cond3]
```

정렬, Aggregation 함수, GroupBy 적용

DataFrame, Series의 정렬 - sort_values()

- by 특정 칼럼 - 해당 칼럼으로 정렬 수행
- ascending = True(디폴트값) 오름차순, False 내림차순
- inplace = False(디폴트값) 원래 데이터 유지, 정렬된 새로운 df 반환

```
titanic_sorted = titanic_df.sort_values(by=['Name'])  # 이름 알파벳 순으로 정렬
```

```
# 내림차순
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Name'], ascending=False)
```

Aggregation 함수 적용

min(), max(), sum(), count()와 같은 aggregation 함수. df에서 바로 이 함수 호출할 경우 모든 칼럼에 해당 aggregation을 적용하게 됨.

```
titanic_df.count()
'''
PassengerId    891
Survived       891
Pclass         891
Name           891
Sex            891
Age            714
SibSp          891
Parch          891
Ticket         891
Fare           891
Cabin          204
Embarked       889
dtype: int64
'''
```

특정 칼럼에 적용

```
titanic_df[['Age','Fare']].mean()
'''
Age     29.699118
Fare    32.204208
dtype: float64
'''
```

groupby() 적용

- by: 대상 칼럼으로 groupby 됨

```
titanic_groupby = titanic_df.groupby(by='Pclass')
print(type(titanic_groupby))  # <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

DF에 groupby( ) 호출로 반환된 결과에 aggregation 함수 호출하면, groupby( ) 대상 칼럼 제외한 모든 칼럼에 해당 aggregation 함수를 적용함

```
titanic_groupby = titanic_df.groupby(by='Pclass').count()
titanic_groupby
'''
PassengerId Survived  Name  Sex Age SibSp Parch Ticket  Fare  Cabin Embarked
Pclass
1 216 216 216 216 186 216 216 216 216 176 214
2 184 184 184 184 173 184 184 184 184 16  184
3 491 491 491 491 355 491 491 491 491 12  491
'''

# 특정 칼럼만 agg 함수 적용 -> 해당 칼럼 필터링 후 agg 함수 적용
titanic_groupby = titanic_df.groupby('Pclass')[['PassengerId','Survived']].count()
```

groupby 함수에 여러개의 agg 함수 적용 → agg 내에 인자로 입력

```
titanic_df.groupby('Pclass')['Age'].agg(['max','min'])
```

여러 개의 칼럼이 서로 다른 agg 함수 적용 → 딕셔너리 형태로 칼럼과 agg 함수 입력

```
agg_format={'Age':'max', 'SibSp':'sum', 'Fare':'mean'}
titanic_df.groupby('Pclass').agg(agg_format)
```

결손 데이터 처리하기

NULL 값은 넘파이의 NaN으로 표시됨. → 다른 값으로 대체해야 함.(null값은 평균, 총합 등의 함수 연산 시 제외됨)

isna()로 결손 데이터 여부 확인 // 결손 데이터 개수 구하기

```
titanic_df.isna().head()
'''
PassengerId Survived  Pclass  Name  Sex Age SibSp Parch Ticket  Fare  Cabin Embarked
0 False False False False False False False False False False True  False
1 False False False False False False False False False False False False
'''

titanic_df.isna().sum()
'''
PassengerId      0
```

```
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
'''
```

fillna( ) 로 Missing 데이터 대체하기

- inplace = True를 통해 실제 데이터 값 변경 가능

```
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
```

```
titanic_df['Age'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())
titanic_df['Embarked'] = titanic_df['Embarked'].fillna('S')
titanic_df.isna().sum()  # 전부 0으로 나옴
```

apply lambda 식으로 데이터 가공

판다스는 apply 함수에 lambda 식을 결합해 DF이나 Series의 레코드별로 데이터를 가공하는 기능을 제공함. 판다스는 칼럼에 일괄적으로 데이터 가공하는 것이 더 빠르나, 복잡한 데이터 가공이 필요할 때 apply lambda 사용함.

lambda

: 파이썬에서 함수형 프로그래밍 지원하게 위해 만들어짐.

```
def get_square(a):
    return a**2

print('3의 제곱은:',get_square(3))
```

↑ 함수는 함수명과 입력 인자를 먼저 선언한 후, 함수 내에서 입력 인자를 가공한 뒤, 결과값을 return과 같은 문법으로 반환해야 함.

lambda는 이러한 함수 선언과 함수 내의 처리를 한 줄의 식으로 쉽게 변환하는 식.

```
lambda_square = lambda x : x**2
print('3의 제곱은:',lambda_square(3))  # 3의 제곱은: 9
```

→ :로 입력 인자와 계산식 분리. 오른쪽 계산식은 반환 값을 의미함.

lambda 식을 이용할 때 여러 개의 값을 입력 인자로 사용 할 경우 → map( ) 함수 결합

```
a=[1,2,3]
squares = map(lambda x : x**2, a)
list(squares)  # [1, 4, 9]
```

판다스 df의 lambda 식은 이러한 lambda 식을 그대로 적용한 것.

```
# 'Name' 칼럼의 문자열 개수를 별도 칼럼인 'Name_len'에 생성
titanic_df['Name_len'] = titanic_df['Name'].apply(lambda x : len(x))
```

```
# 나이가 15세 미만 -> child, 그렇지 않으면 -> adult
titanic_df['Child_Adult'] = titanic_df['Age'].apply(lambda x : 'Child' if x<=15 else 'Adul
t')
```

lambda if else 지원하는데, 주의할 점 → if 절의 경우 if 식보다 반환 값을 먼저 기술 해야 함! : 오른쪽에 반환 값이 있어야 하기 때문. 또한, else만 지원하고 else if는 지원x(→ 이 경우 else 절에 ( )를 내포해 ( ) 내에서 다시 if else 적용)

```
titanic_df['Age_cat'] = taitanic_df['Age'].apply(lambda x : 'Child' if x<=15 else('Adult'
 if x<=60 else 'Elderly'))
```

else if가 많이 나와야 하느 경우 아예 별도의 함수 만드는게 더 나음.

```
# 나이에 따라 세분화된 분류를 수행하는 함수 생성.
def get_category(age):
    cat = ''
    if age <= 5: cat = 'Baby'
    elif age <= 12: cat = 'Child'
    elif age <= 18: cat = 'Teenager'
```

```
    elif age <= 25: cat = 'Student'
    elif age <= 35: cat = 'Young Adult'
    elif age <= 60: cat = 'Adult'
    else : cat = 'Elderly'

    return cat

# lambda 식에 위에서 생성한 get_category( ) 함수를 반환값으로 지정.
# get_category(X)는 입력값으로 'Age' 컬럼 값을 받아서 해당하는 cat 반환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : get_category(x))
titanic_df[['Age','Age_cat']].head()
```

pandas 정리

`pd.read_csv()`	`read_csv(파일 경로명)`
`df.head()`	맨 앞 5개의 로우 반환(디폴트값=5)
`df.shape`	DataFrame의 행과 열을 튜플 형태로 반환
`df.info()`	총 데이터 건수, 칼럼별 데이터 타입, Null 데이터 개수, 데이터 분포도 등의 메타 데이터
`df.describe()`	숫자형 칼럼의 분포도 조사(분포도, 평균값, 최댓값, 최솟값 등) → 개략적 분포 파악 count: Not Null인 데이터 건수, mean: 전체 데이터 평균값, std: 표준편차...
`df['column 명'].value_counts()`	series 형태로 특정 칼럼 데이터 세트 반환, 해당 칼럼값의 유형과 검수 확인 가능 건수가 많은 순서로 정렬되어 값 반환
`pd.DataFrame(list/array, columns=)`	list나 ndarray → df로 변환
`df.values`	df → ndarray 타입으로 변환
`df['새로운 칼럼명']=0`	새로운 칼럼 추가, 값 할당
`df.drop()`	axis=0(행), axis=1(열) inplace = False (디폴트, 원본 df 유지) 여러개 삭제 → 리스트 형식
`df.index`	인덱스 객체 추출
`df.reset_index`	새롭게 인덱스 연속 숫자 형으로 할당 & 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가
`df.iloc[0,0]`	위치 기반 인덱싱
`df.loc['one','Name']`	명칭 기반 인덱싱
`df[df['Age'] > 60 ]`	불린 인덱싱
`df.sort_values`	정렬(해당 칼럼 기준으로 오름차순/내림차순 정렬 수행 가능)
agg - `df.count()`	

agg - `df.mean()`	
`df.groupby(by='Pclass')`	해당 칼럼을 기준으로 ...
`df.isna()`	t/f로 nan 값 여부 알려줌
`df.isna().sum()`	칼럼 별 결손 데이터 개수 알려줌
`df['Cabin'].fillna('C')`	결손 값 채우기

05. 정리

ML 알고리즘이 차지하는 비중보다 데이터를 전처리하고 적절한 피처를 가공/추출하는 부분이 훨씬 많은 비중을 차지하게 됨. 넘파이, 판다스, 맷플롯립/시본과 같이 파이썬 머신러닝 생태계 이루고 있는 다양한 패키지에 대한 이해 매우 중요함.