01. 머신러닝의 개념

✔️ 머신러닝의 개념

: 애플리케이션을 수정하지 않고도 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법

사용 예시

1. 모든 조건과 다양한 환경 변수, 규칙을 반영하여 금융거래 사기 적발을 하는 프로그램

   ⇒ 매우 복잡한 조건들을 반영하여 정확히 예측하기 어려움

   ⇒ 업무적으로 복잡한 조건/규칙들이 다양한 형태로 결합하고 시시각각 변하면서 도저히 소프트웨어 코드로 로직을 구성하여 이를 관통하는 일정한 패턴을 찾기 어려운 경우에 머신러닝은 훌륭한 솔루션을 제공

2. 문맥에 따라 스팸메일을 분류하는 프로그램

   ⇒ 인간의 언어에서 패턴을 규정하기 어려움

   ⇒ 머신러닝은 데이터를 기반으로 숨겨진 패턴을 인지해 문제를 해결

✅ 데이터를 기반으로통계적인 신뢰도를 강화하고 예측 오류를 최소화하기 위한 다양한 수학적 기법을 적용해 데이터 내의 패턴을 스스로 인지하고 신뢰도 있는 예측 결과를 도출

성과

1. 데이터에 감춰진 새로운 의미와 인사이트를 발굴해 놀랄 만한 이익으로 연결
2. 데이터마이닝, 영상인식, 음성인식, 자연어 처리에서 개발자가 데이터나 업무 로직을 직접 감안한 프로그램을 만들 경우 난이도와 개발 복잡도가 너무 높아질 수밖에 없는 분야에서 머신러닝이 급속하게 발전을 이루고 있음

✔️ 머신러닝의 분류

지도학습(Supervised Learning)	분류	회귀
비지도학습(Un-Supervised Learning)	클러스터링(군집화)	차원 축소
강화학습(Reinforcement Learning)		

✔️ 데이터 전쟁

1. 머신러닝은 데이터에 매우 의존적임
2. 좋은 품질의 데이터를 갖추지 못한다면 머신러닝의 수행 결과도 좋을 수 없음

⇒ ✅ 데이터를 이해하고 효율적으로 가공, 처리, 추출해내 최적의 데이터를 기반으로 알고리즘을 구동할 수 있도록 준비하는 능력이 더 중요할 수 있음.

✅ 다양하고 광대한 데이터를 기반으로 만들어진 머신러닝 모델은 더 좋은 품질을 약속할 수 있음

✔️ 파이썬과 R기반의 머신러닝 비교

R	파이썬
통계 전용 프로그램 언어	개발 전문 프로그램 언어
다양하고 많은 통계 패키지	유연한 아키텍처, 다양한 라이브러리

파이썬의 장점

1. 쉽고 뛰어난 개발 생산성 ⇒ 전 세계 개발자들이 선호
2. 오픈 소스 계열의 전폭적 지원, 많은 라이브러리 ⇒ 높은 생산성
3. 속도는 느리지만 뛰어난 확장성, 유연성, 호환성 ⇒ 다양한 영역에서 사용
4. 머신러닝 애플리케이션과 결합한 다양한 애플리케이션 개발 가능
5. 다양한 기업 환경으로의 확산이 가능
6. 딥러닝 프레임워크인 TensorFlow, Keras, PyThorch 등에서 파이썬 우선 정책으로 파이썬을 지원

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

✔️ 파이썬 기반의 머신러닝을 익히기 위해 필요한 패키지

기능	패키지 이름	설명
머신러닝	사이킷런(Scikit-Learn)	데이터 마이닝 기반의 머신러닝에서 독보적 위치 차지
행렬/선형대수/통계	넘파이(NumPy), 사이파이(SciPy)	행렬 기반의 데이터 처리에 특화, 사이킷런 역시 사이파이 패키지의 도움을 받아 구축된 여러 패키지 가지고 있음
데이터 핸들링	판다스	2차원 데이터 처리에 특화, 넘파이보다 훨씬 편리하게 데이터 처리할 수 있는 많은 기능 제공, 맷플롯립(Matplotlib)을 호출해 쉽게 시각화 기능 지원
시각화	맷플롯립(Matplotlib)	오랜 기간 동안 대표적인 시각화 라이브러리, 세분화된 API로 익히기 어려움, 투박한 디자인, 코드가 길어서 비효율적
	시본(Seaborn)	맷플롯립을 보완하는 시각화 패키지, 맷플롯립을 기반으로 함 ⇒ 맷플롯립 사용법 어느정도 알아야함 판다스와의 쉬운 연동, 함축적인 API, 다양한 그래프/차트 제공

- 주피터 노트북(Jupyter Notebook)

  : 아이파이썬(대화형 파이썬 툴) 지원 툴. 학생들이 필기하듯이 중요 코드 단위로 설명을 적고 코드를 수행해 그 결과를 볼 수 있게 만들어서 직관적으로 어떤 코드가 어떤 역할을 하는지 매우 쉽게 이해할 수 있도록 지원

✔️ 파이썬 머신러닝을 위한 S/W 설치

1. Anaconda 설치 → 파이썬 머신러닝 패키지 함께 설치됨
2. Microsoft Visual Studio Build Tools 2015 이상 버전 설치

✔️ 넘파이와 판다스의 중요성

1. 머신러닝 애플리케이션을 파이썬으로 개발할 때 대부분의 코드는 사이킷런의 머신러닝 알고리즘에 입력하기 위한 데이터의 추출/가공/변형, 원하는 차원 배열로의 변환을 포함해 머신러닝 알고리즘 처리 결과에 대한 다양한 가공으로 구성 ⇒ 데이터 처리 부분은 대부분 넘파이와 판다스의 몫
2. 사이킷런이 넘파이 기반에서 작성 됨 ⇒ 넘파이의 기본 프레임워크를 이해하지 못하면 사이킷런 역시 실제 구현에서 어려움

⇒ 넘파이와 판다스에 대한 기본 프레임워크와 중요 API만 습득하고, 일단 코드와 부딪혀 가면서 모르는 API에 대해서는 인터넷을 통해 체득해야 함

03. 넘파이

✔️ 넘파이의 개념

넘파이(NUMerical PYthon)

: 파이썬의 선형대수 프로그램을 쉽게 만들 수 있도록 지원하는 대표적인 패키지

장점

1. 루프를 사용하지 않고 대량의 데이터의 배열 연산을 가능하게 함 → 빠른 배열 연산 속도를 보장

   ⇒ 빠른 계산 능력은 대량 데이터 기반의 과학과 공학에서 중요하므로 파이썬 기반의 많은 과학/공학 패키지는 넘파이에 의존하고 있음

2. 저수준 언어(C/C++ 등)의 호환 API 제공

   ⇒ 수행 성능이 매우 중요한 부분은 저수준 언어로 작성하고 이를 넘파이에서 호출하는 방식 사용

   ⇒ 넘파이의 빠른 연산 + 저수준 언어의 빠른 수행 성능(TensorFlow도 이 방식으로 작성됨)

3. 다양한 데이터 핸들링 기능 제공

단점

1. 편의성과 다양한 API 지원 측면에서 아쉬움
2. 일반적으로 데이터는 2차원 형태이므로 판다스의 편리성에 못미침

중요한 이유

1. 많은 머신러닝 알고리즘이 넘파이 기반으로 작성되어 있음
2. 머신러닝 알고리즘의 입력 데이터와 출력 데이터를 넘파이 배열 타입으로 사용
3. 넘파이가 배열을 다루는 기본 방식을 이해하는 것은 다른 데이터 핸들링 패키지(판다스 등)을 이해하는 데에 많은 도움이 됨

✔️ 넘파이 ndarray 개요

- **넘파이 모듈 임포트**

```
import numpy as np
```

- **ndarray**

: 넘파이 기반 데이터 타입

⇒ ndarray를 이용하여 넘파이에서 다차원 배열을 쉽게 생성하고 다양한 연산을 수행할 수 있음

- **넘파이 array( ) 함수**

: 파이썬의 리스트와 같은 다양한 인자를 입력받아서 ndarray로 변환하는 기능을 수행

- **shape 변수**

: ndarray의 크기(행과 열의 수)를 튜플 형태로 가지고 있음.

⇒ ndarray 배열의 차원까지 알 수 있음

- **np.array( )**

  : ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환

- **ndarray.shape**

  : ndarray의 차원과 크기를 튜플 형태로 반환

  ⇒ (로우, 칼럼)

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

```
# 1차원 array로 3개의 데이터를 가지고 있음

array2 = np.array([[1,2,3],
                   [2,3,4]])
print('array2 type: ', type(array2))
print('array2 array 형태: ', array2.shape)
# array2 type:  <class 'numpy.ndarray'>
# array2 array 형태:  (2, 3)
# 2차원의 array로, 2개의 로우와 3개의 칼럼으로 구성되어 6개의 데이터를

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

- **ndarray.ndim**

  : ndarray의 차원 반환

  ```
  print('array1: {:0}차원, array2: {:1}차원, array3: {:2}차원'
  # array1: 1차원, array2: 2차원, array3:  2차원
  ```

  리스트 [ ]는 1 차원이고 [[ ]]는 2차원 같은 형태이므로 array3은 2차원임

✔️ ndarray의 데이터 타입

- **ndarray내의 데이터값**

  1. 숫자 값, 문자열 값, 불 값 등 모두 가능
  2. 숫자형은 int형(8bit, 16bit, 32bit), unsigned int형(16bit, 32bit, 64bit, 128bit), complex 타입(더 큰 숫자나 정밀도를 위해)

- **ndarray내의 데이터 타입**

  : 연산의 특성상 같은 데이터 타입만 가능

  ⇒ dtype 속성으로 데이터 타입 확인

```
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
```

- **다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변환하는 경우**

  : 데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용

  eg) int형과 float형이 섞여 있는 리스트를 ndarray로 변환 → float형으로 적용

- **astype( ) 메서드**

  : ndarray 내 데이터값의 타입 변경, 메모리 절약해야 할 때 이용

  ⇒ 인자로 원하는 타입을 문자열로 지정

```
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

: 특정 크기와 차원을 가진 ndarry에 연속값/0/1로 초기화해 생성해야하는 경우

- **arange( )**

  : array를 range( )로 표현

  ⇒ 0~(함수인자값 - 1)을 순차적으로 ndarray의 데이터 값으로 변환

  ⇒ range( )처럼 start 값 부여 가능

```
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( )**

  : 함수 인자로 튜플 형태의 shape값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray로 반환

- **ones( )**

  : 함수 인자로 튜플 형태의 shape값을 입력하면 모든 값을 1로 채운 해당 shape를 가진 ndarray로 반환

  ⇒ 함수 인자로 dtype을 정해주지 않으면 default로 float64 형의 데이터로 채움

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

```
'''
[[0 0]
 [0 0]
 [0 0]]
int32 (3, 2)
'''

one_array = np.ones((3,2))
print(one_array)
print(one_array.dtype, one_array.shape)

'''
[[1. 1.]
 [1. 1.]
 [1. 1.]]
float64 (3, 2)
'''
```

✔️ ndarray의 차원과 크기를 변경하는 reshape( )

- reshape( )

  : ndarray를 특정 차원 및 크기로 변환, 변환을 원하는 크기를 함수 인자로 부여

  ⇒ 지정된 사이즈로 변경이 불가능하면 오류 발생

- 인자로 -1을 적용하는 경우

  : -1은 자동으로 적절한 크기를 계산하라는 지시

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

array2 = array1.reshape(-1, 5)
print('array2 shape: ', array2.shape)
# array2 shape:  (2, 5)
```

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

- reshape(-1, 1)

  : 원본 ndarray가 어떤 형태라도 2차원이고 여러 개의 로우를 가지되 반드시 1개의 칼럼을 가진 ndarray로 변환됨을 보장

```
array1 = np.arange(8)
array3d = array1.reshape((2,2,2))
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:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
array5 shape:  (8, 1)
'''

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

✔️ 넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱 (Indexing)

단일 값 추출

- **1차원 ndarray에서 한 개의 데이터 추출**

  : 해당하는 위치의 인덱스 값을 [ ]안에 입력, 인덱스를 이용해 데이터값 수정도 가능

```
# 1부터 9까지의 1차원 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 데이터값
value = array1[2]
print('value: ', value)
print(type(value))
'''
value:  3
<class 'numpy.int64'>
'''

print('맨 뒤의 값: ', array1[-1], '맨 뒤에서 두 번째 값: ', arra
# 맨 뒤의 값:  9 맨 뒤에서 두 번째 값:  8

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

- **다차원 ndarray에서 한 개의 데이터 추출**

  : 콤마로 분리된 로우와 칼럼 위치의 인덱스를 통해 접근

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

print('(row=0, com=0) index 가리키는 값: ', array2d[0,0])
print('(row=0, com=1) index 가리키는 값: ', array2d[0,1])
print('(row=1, com=0) index 가리키는 값: ', array2d[1,0])
print('(row=2, com=2) index 가리키는 값: ', array2d[2,2])

'''
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(row=0, com=0) index 가리키는 값:  1
(row=0, com=1) index 가리키는 값:  2
(row=1, com=0) index 가리키는 값:  4
(row=2, com=2) index 가리키는 값:  9
'''
```

✅ 넘파이의 다차원 ndarray는 axis 구분을 가짐

axis 0이 로우 방향 축, axis 1이 칼럼 방향 축

[row=0, col=1] == [axis0=0, axis1=1]

슬라이싱

: 연속된 인덱스상의 ndarray를 추출

- 1차원 ndarray에서 슬라이싱

  : [시작인덱스:종료인덱스] ⇒ 시작인덱스 ~ (종료인덱스-1)까지 ndarray로 추출

```
array1 = np.arange(start=1, stop=10)
array3 = array1[0:3]
print(array3)
print(type(array3))
```

```
'''
[1 2 3]
<class 'numpy.ndarray'>
'''
```

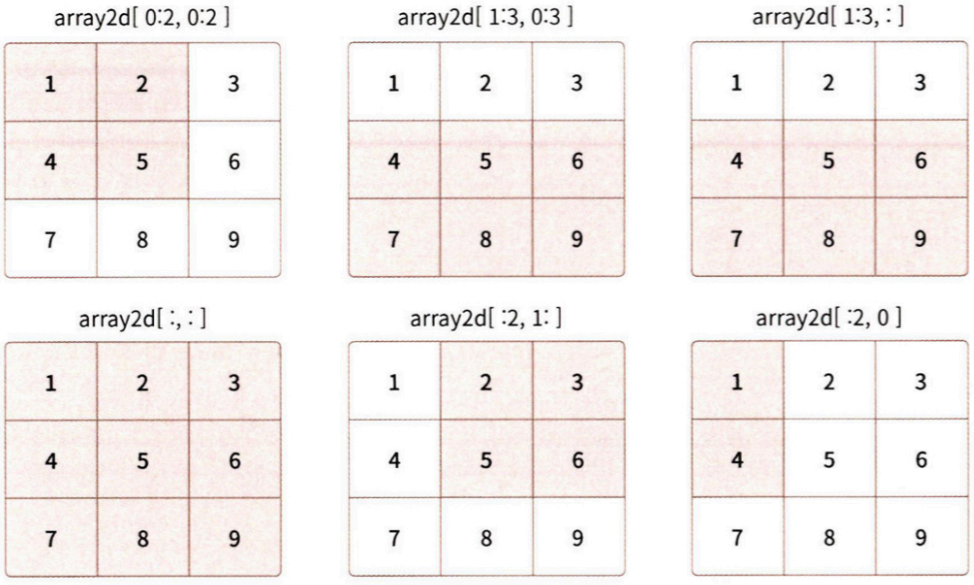
- 인덱스 생략하는 경우
  1. 시작 인덱스 생략 ⇒ 맨 처음 인덱스인 0으로 간주
  2. 종료 인덱스 생략 ⇒ 맨 마지막 인덱스로 간주
  3. 시작&종료 인덱스 생략 ⇒ 맨 처음&마지막 인덱스로 간주

- 다차원 ndarray에서 슬라이싱

  : 콤마로 분리된 로우와 칼럼 위치의 인덱스를 통해 접근

  - 인덱스 생략하는 경우

    : 1차원과 동일, 뒤에 오는 인덱스를 없애면 1차원 ndarray를 반환



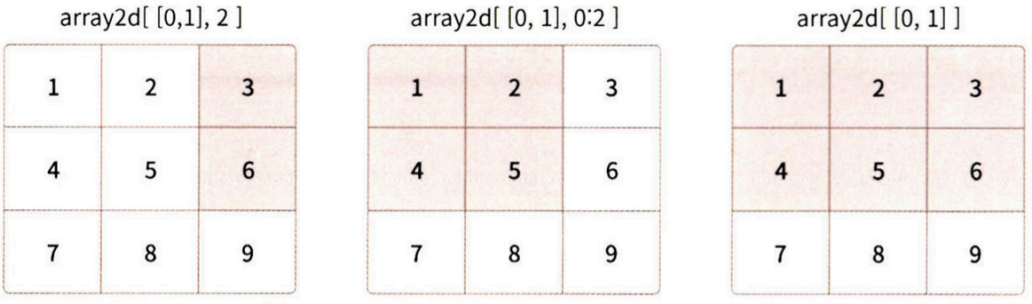

```
print(array2d[0])
print(array2d[1])
print('array2d[0] shape: ', array2d[0].shape, 'array2d[1]

'''
```

```
[1 2 3]
[4 5 6]
array2d[0] shape:  (3,) array2d[1] shape:  (3,)
'''
```

펜시인덱싱

: 리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당하는 ndarray를 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[0:2, 0:2] => ', array4.tolist())
# array2d[0:2, 0:2] =>  [[1, 2], [4, 5]]

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



불린 인덱싱

: 조건 필터링과 검색을 할 수 있는 인덱싱

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

# [ ] 안에 array1d > 5 Boolean indexing을 적용
array3 = array1d[array1d > 5]
print('array1d > 5 불린 인덱싱 결과 값: ', array3)
# array1d > 5 불린 인덱싱 결과 값:  [6 7 8 9]
```

- **동작 방식**

    1. array1d > 5와 같이 필터링 조건은 [ ] 안에 기재
    2. False 값은 무시하고 True 값에 해당하는 인덱스값만 저장
       (True 값 자체가 아니고 True값을 가진 인덱스를 저장)
    3. 저장된 인덱스 데이터 세트로 ndarray 조회

✔️ 행렬의 정렬 - sort( )와 argsort( )

행렬 정렬

- **np.sort( )**

  : 넘파이에서 sort( )를 호출하는 방식

  ⇒ 원 행렬은 그대로 유지한 채 원 행렬의 정렬된 행렬을 반환

- **ndarray.sort( )**

  : 행렬 자체에서 sort( )를 호출하는 방식

  ⇒ 원 행렬 자체를 정렬한 형태로 변환

	np.sort( )	ndarray.sort( )
원본 행렬	[3, 1, 9, 5]	[3, 1, 9, 5]

	np.sort( )	ndarray.sort( )
호출 후 반환 행렬	[1, 3, 5, 9]	None
호출 후 원본 행렬	[3, 1, 9, 5]	[1, 3, 5, 9]

- **내림차순**

  : [::-1] 사용

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

- **다차원 행렬의 정렬**

  : axis 축 값 설정을 통해 로우 방향, 또는 칼럼 방향으로 정렬을 수행

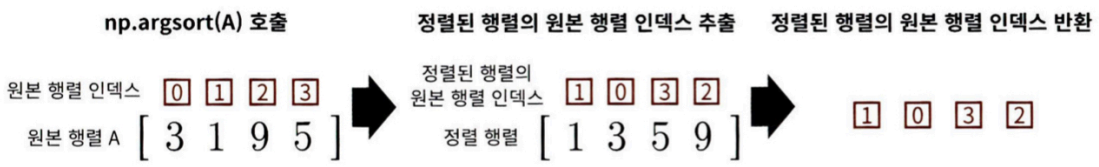
```
array2d = np.array([[8,12], [7,1]])

sort_array2d_axis0 = np.sort(array2d, axis=0)
print('로우 방향으로 정렬: \n', sort_array2d_axis0)
'''
로우 방향으로 정렬:
 [[ 7  1]
 [ 8 12]]
 '''

sort_array2d_axis1 = np.sort(array2d, axis=1)
print('칼럼 방향으로 정렬: \n', sort_array2d_axis1)
'''
칼럼 방향으로 정렬:
 [[ 8 12]
 [ 1  7]]
 '''
```

정렬된 행렬의 인덱스를 반환하기

- **np.argsort( )**

  : 원본 행렬이 정렬되었을 때 기존 원본 행렬의 원소에 대한 인덱스를 필요로 할 때 사용

  ⇒ 정렬 행렬의 원본 행렬 인덱스를 ndarray형으로 반환



- **활용**

  : 넘파이는 RDBMS의 TABLE 칼럼이나 판다스 DataFrame 칼럼과 같은 메다 테이터를 가질 수 없음

  → 실제 값과 그 값이 뜻하는 메타 데이터를 별도의 ndarray로 각각 가져야만 함

```
# 학생별 시험 성적을 데이터로 표현하는 방법

name_array = np.array(['John', 'Mike', 'Sarah', 'Kate', 'S
score_array = np.array([78, 95, 84, 98, 88])

# 시험 성적 순으로 학생 이름 출력
sort_indices_asc = np.argsort(score_array)
print('성적 오름차순 정렬 시 score_array의 인덱스: ', sort_indic
print('성적 오름차순으로 name_array의 이름 출력: ', name_array[s
# 펜시 인덱스 적용

'''
성적 오름차순 정렬 시 score_array의 인덱스:  [0 2 4 1 3]
성적 오름차순으로 name_array의 이름 출력:  ['John' 'Sarah' 'Samu
'''
```

✔️ 선형대수 연산 - 행렬 내적과 전치 행렬 구하기

- **행렬 내적(행렬 곱): np.dot( )**

  행렬곱

  : 두 행렬 A와 B의 내적은 왼쪽 행렬의 로우(행)과 오른쪽 행렬의 칼럼(열)의 원소들을 순차적으로 곱한 뒤 그 결과를 모두 더한 값

  ⇒ 왼쪽 행렬의 열 개수와 오른쪽 행렬의 행 개수가 동일해야 내적 연산이 가능

```
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: transpose( )**

  전치행렬

  : 원 행렬에서 행과 열 위치를 교환한 원소로 구성한 행렬

  ⇒ A 행렬의 전치행렬 = A^T

```
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]]
 '''
```

04. 데이터 핸들링 - 판다스

✔ 판다스 소개

1. 월스트리트 금융회사의 분석 전문가인 Wes McKinney가 개발
2. 데이터 처리를 위해 존재하는 가장 인기있는 라이브러리
3. 행과 열로 이뤄진 2차원 데이터를 효율적으로 가공/처리할 수 있는 다양하고 훌륭한 기능 제공
4. RDBMS의 SQL이나 엑셀 시트에 버금가는 고수준 API 제공
5. 파이썬의 리스트, 컬렉션, 넘파이 등 내부 데이터와 CSV 등의 파일을 쉽게 DataFrame으로 변경해 데이터의 가공/분석을 편리하게 수행할 수 있게 해줌

핵심개체: DataFrame

- Index: RDBMS의 PK처럼 개별 데이터를 고유하게 식별하는 Key 값
- DataFrame: 2차원 데이터를 담는 데이터 구조체, Index를 Key 값으로 가짐, 칼럼이 여러개인 구조체(여러개의 Series로 이루어짐)
- Series: index를 key 값으로 가짐, 칼럼이 하나인 구조체

✔ 판다스 시작 - 파일을 DataFrame으로 로딩, 기본 API

- **판다스 모듈 임포트**

```
import pandas as pd
```

- **파일을 DataFrame으로 로딩하는 API**

read_csv( )	csv(칼럼을 콤마로 구분한 파일 포맷) 파일 포맷 변환

read_table( )	칼럼을 tab(\t)로 구분한 파일 포맷의 파일 포맷 변환
read_fwf( )	고정 길이 기반의 칼럼 포맷을 로딩

read_csv( )

- read_csv( )의 인자인 sep에 구분 문자를 입력해서 설정 가능

  ⇒ read_csv('파일명', sep='\t')

- read_csv(filepath, sep=' ', ...)에서 filepath는 필수로 입력해야함
- 별다른 파라미터 지정이 없으면 파일의 맨 처음 로우를 칼럼명으로 인지하고 칼럼으로 변환
- 모든 DataFrame 내의 데이터는 생성되는 순간 고유의 Index 값을 가지게 됨

- **DataFrame.head( )**

  : 맨 앞 n개의 로우를 반환. Default는 5개

- **shape 변수**

  : DataFrame의 행과 열을 튜플 형태로 반환

```
print('DataFrame 크기: ', titanic_df.shape)
# DataFrame 크기:  (891, 12)
```

- **info( )**

  : 총 데이터 건수와 데이터 타입, Null 건수를 알 수 있음

- **describe( )**

  : 칼럼별 숫자형 데이터값의 n-percentile 분포도, 평균값, 최댓값, 최솟값을 나타냄. 오직 숫자형 칼럼의 분포도만 조사

  ⇒ 숫자 형 칼럼에 대한 개략적인 데이터 분포도를 확인할 수 있음

- **value_counts( )**

  : 칼럼값의 유형과 건수를 확인할 수 있음. Series 객체에서만 정의

  ⇒ 데이터의 분포도를 확인하는 데 매우 유용한 함수.

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

- **DataFrame 내의 특정 칼럼 데이터 세트 반환**

  : DataFrame의 [ ] 연산자 내부에 칼럼명 입력 → 해당 칼럼에 해당하는 Series 객체 반환

  ✅ Series

  - 단 하나의 칼럼으로 구성된 데이터 세트
  - 인덱스 + 데이터 값으로 구성
  - 인덱스는 순차값으로 지정될 수도 있고 고유성이 보장된다면 의미 있는 데이터 값도 할당 가능
  - 인덱스는 DataFrame, Series가 만들어진 후에도 변경 가능

✔️ DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- DataFrame ↔ 파이썬의 리스트 , 딕셔너리, 넘파이 ndarray

- 사이킷런의 많은 API는 DataFrame을 인지로 입력받을 수 있지만, 기본적으로 넘파이 ndarray를 입력인자로 사용하는 경우가 대부분

  ⇒ DataFrame과 넘파이 ndarray 상호 간의 변환이 매우 빈번하게 발생

넘파이 ndarray, 리스트, 딕셔너리 → DataFrame

✅ DataFrame은 ndarray와 다르게 칼럼명을 가지고 있음

✅ 일반적으로 DataFrame으로 변환할 때 칼럼명을 지정(지정하지 않으면 자동으로 할당)

✅ data(리스트/딕셔너리/ndarray 입력) + columns(칼럼명 리스트 입력) → DataFrame 생성

✅ 2차원 이하의 데이터들만 변환 가능

- **1차원 형태 데이터 변환**

  : 칼럼명이 한 개만 필요

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

df_list1 = pd.DataFrame(list1, columns=col_name1)
print('1차원 리스트로 만든 DataFrame: \n', df_list1)
# 넘파이 ndarray를 이용해 DataFrame 생성

df_array1 = pd.DataFrame(array1, columns=col_name1)
print('1차원 ndarray로 만든 DataFrame: \n', df_array1)

'''
array1 shape:  (3,)
1차원 리스트로 만든 DataFrame:
    col1
0     1
```

```
1     2
2     3
1차원 ndarray로 만든 DataFrame:
    col1
0     1
1     2
2     3
'''
```

- **2차원 형태 데이터 변환**

  : 열의 수에 맞춰서 칼럼 수 필요

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

# 2행x3열 형태의 리스트와 ndarray 생성한 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
'''
```

- **딕셔너리 변환**

  : 키 → 칼럼명(문자열), 값 → 키에 해당하는 칼럼 데이터(리스트/ndarray)

```
# key는 문자열 칼럼명으로 매핑, Value는 리스트 형(또는 ndarray) 칼
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
'''
```

DataFrame → 넘파이 ndarray, 리스트, 딕셔너리

- **DataFrame → ndarray**

  : 많은 머신러닝 패키지가 기본 데이터 형으로 넘파이 ndarray를 사용하기 때문에 빈번하게 변환

  ⇒ DataFrame 객체의 values를 이용해 쉽게 변환 가능

```
# DataFrame을 ndarray로 변환
array3 = df_dict.values
print('df_dict.values 타입: ', type(array3), 'df_dict.value
print(array3)

'''
df_dict.values 타입:  <class 'numpy.ndarray'> df_dict.value
[[ 1  2  3]
 [11 22 33]]
'''
```

- **DataFrame → 리스트**

  : values로 얻은 ndarray에 tolist( ) 호출

- **DataFrame → 딕셔너리**

  : DataFrame 객체의 to_dict( )메서드 호출. 인자로 'list' 입력하면 딕셔너리의 값이 리스트형으로 반환

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

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

✔️ DataFrame의 칼럼 데이터 세트 생성과 수정

- **칼럼 데이터 생성**

  : [ ] 연산자 이용, 기존 칼럼 Series의 데이터를 이용하는 것도 가능

```
titanic_df['Age_0'] = 0

titanic_df['Age_by_10'] = titanic_df['Age']*10
titanic_df['Family_No'] = titanic_df['SibSp'] + titanic_df
```

- **칼럼 데이터 수정**

  : 업데이트를 원하는 칼럼 Series를 DataFrame [ ] 내에 칼럼 명으로 입력한 뒤에 값을 할당

```
titanic_df['Age_by_10'] = titanic_df['Age_by_10'] + 100
```

✔️ DataFrame 데이터 삭제

drop( )

DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

- **axis**

  : DataFrame의 로우를 삭제할 때는 axis=0, 칼럼을 삭제할 때는 axis=1

  ⇒ 주로 칼럼을 드롭. 로우는 이상치 데이터를 삭제하는 경우에 주로 사용

- **labels**

  : 삭제할 칼럼 명/인덱스

  ⇒ axis =0 → labels: 인덱스 값

  DataFrame.drop(삭제할 인덱스 번호, axis=0)

  ⇒ axis =1 → labels: 칼럼 명

  DataFrame.drop('칼럼명', axis=1)

- **inplace**

  : inplace=False면 자신의 DataFrame의 데이터는 삭제하지 않음, inplace=True면 자신의 DataFrame의 데이터 삭제

  ⇒ inplace = true일 때 반환 값을 자기 자신의 DataFrame 객체로 할당하면 안됨 (None 반환)

- **여러 개의 칼럼 삭제**

  : 리스트 형태로 삭제하고자 하는 칼럼 명을 입력해 labels 파라미터로 입력

```
drop_result = titanic_df.drop(['Age_0', 'Age_by_10', 'Fami
```

✔️ Index 객체

- **Index**

  : DataFrame, Series의 레코드를 고유하게 식별하는 개체

- **Index 개체 추출**

  : DataFrame.index, Series.index 속성 이용

  ⇒ 넘파이 1차원 ndarray로 볼 수 있음

- **Index 속성**

  1. ndarray와 유사하게 단일 값 반환 및 슬라이싱 가능함

```
print(indexes.values.shape)
print(indexes[:5].values)
print(indexes.values[:5])
print(indexes[6])

'''
(891,)
[0 1 2 3 4]
[0 1 2 3 4]
6
'''
```

2. 한 번 만들어진 Index 개체는 함부로 변경할 수 없음
3. Series 객체에 연산 함수를 적용할 때 Index는 연산에서 제외됨. 식별용으로만 사용

- **reset_index( )**

: 새롭게 인덱스를 연속 숫자 형으로 할당, 기존 인덱스는 'index'라는 새로운 칼럼명으로 추가

⇒ 인덱스가 연속된 숫자형 데이터가 아닐 경우에 다시 이를 연속 int 숫자형 데이터로 만들 때 주로 사용

✅ Series에 reset_index( )를 적용하면 DataFrame이 반환

✔️ 데이터 셀렉션 및 필터링

DataFrame의 [ ] 연산자

- **넘파이와의 차이**
  - 넘파이의 [ ] 연산자: 행의 위, 열의 위치, 슬라이싱 범위 등을 지정해 데이터 가져옴
  - DataFrame의 [ ] 연산자: 칼럼 명 문자/칼럼 명 리스트 객체/인덱스로 변환 가능한 표현식이 들어갈 수 있음

    ⇒ 칼럼만 지정할 수 있는 '칼럼 지정 연산자', 칼럼명이 아니면 오류 발생

```
print('단일 칼럼 데이터 추출:\n', titanic_df['Pclass'].head
print('\n여러 칼럼들의 데이터 추출:\n', titanic_df[['Survive
```

- **DataFrame[ ]**
  - 판다스의 인덱스 형태로 변환 가능한 표현식은 [ ]내에 입력할 수 있음

    ⇒ 사용 자제

```
titanic_df[0:2]
```

  - 불린 인덱싱 표현 가능

```
titanic_df[titanic_df['Pclass'] == 3].head(3)
```

DataFrame ix[ ] 연산자

- **칼럼 명칭 기반 인덱싱**

  : ix[인덱스값, 칼럼명]

  eg) titanic_df.ix[0,2]

- **칼럼 위치 기반 인덱싱**

  : ix[인덱스 로우, 인덱스 칼럼]

  eg) titanic_df.ix[0, 'Pclass']

- 넘파이 ndarray의 [ ] 연산자와 같이 단일 지정, 슬라이싱, 불린 인덱싱, 펜시 인덱싱 모두 가능
- 행 위치에 적용되는 인덱스값과 위치 기반 인덱싱이 integer형일 때 코드 작성에 혼선 초래

  → ix[ ] 사용하지 않음, iloc[ ]과 loc[ ] 등장

ix[ ]의 명칭 기반 인덱싱과 위치 기반 인덱싱의 구분

- **명칭 기반 인덱싱**

  : 칼럼의 명칭을 기반으로 위치를 지정하는 방식

- **위치 기반 인덱싱**

  : 0을 출발점으로 하는 가로축, 세로축 좌표 기반의 행과 열 위치를 기반으로 데이터를 지정

  ⇒ 정수 입력

- DataFrame의 인덱스값은 명칭 기반 인덱싱이라고 간주
- ix[ ] 사용하지 않음, iloc[ ]과 loc[ ] 등장

DataFrame iloc[ ] 연산자

- **위치 기반 인덱싱만 허용**
- 행과 열 값으로 integer 또는 integer형의 슬라이싱, 팬시 리스트 값을 입력
- 위치 인덱싱이 아닌 명칭/문자열 인덱스를 위치에 입력하면 오류 발생

- 명확한 위치 기반 인덱싱이 사용되어야 하는 제약으로 인해 불린 인덱싱은 제공하지 않음

  eg) data_df.iloc[0, 0]

DataFrame loc[ ] 연산자

- **명칭 기반으로 데이터 추출**
- 행 위치에는 DataFrame index 값, 열 위치에는 칼럼 명 입력 ⇒ `DataFrame.loc['index 값', '칼럼 명']`
- 정수형 명칭 입력 가능 ⇒ 주의

  loc[ ], ix[ ]에 슬라이싱 적용하는 경우

  : (종료값-1)까지가 아닌 종료값까지 포함

  ⇒ 명칭은 숫자 형이 아닌 수 있기 때문에 -1을 할 수 없음

  ⇒ ix[ ]에 위치 기반 인덱싱이 슬라이싱되면 (종료값-1)까지 포함

```
print('명칭 기반 loc slicing\n', data_df.loc['one':'two', 'Name
print(data_df_reset.loc[1:2, 'Name'])
```

불린 인덱싱

- 처음부터 가져올 값을 조건으로 ix[ ]내에 입력하면 자동으로 원하는 값을 필터링

  ⇒ 자주 사용

- [ ], ix[ ], loc[ ]에서 지원

  ⇒ iloc[ ]은 정수형 값이 아닌 불린 값에 대해서는 지원하지 않음

  ```
  titanic_boolean = titanic_df[titanic_df['Age'] > 60]
  ```

- 반환된 객체의 타입은 DataFrame

  ⇒ [ ]연산자에 칼럼 명을 입력해서 원하는 칼럼 명만 별도로 추출할 수 있음

```
titanic_df[titanic_df['Age'] > 60][['Name', 'Age']].head(3
```

- loc[ ]이용해도 동일

```
titanic_df.loc[titanic_df['Age'] > 60][['Name', 'Age']]
```

- 복합 조건 연산자 사용 가능

```
titanic_df[(titanic_df['Age'] > 60) & (titanic_df['Pcla
```

- 개별 조건을 변수에 할당하고 이들 변수를 결합해서 수행 가능

```
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( )

- **주요 입력 파라미터 by, ascending, inplace**

by	특정 칼럼 입력	해당 칼럼으로 정렬을 수행, 리스트 형식으로 칼럼 입력하면 여러 개의 칼럼으로 정렬
ascending	= True(default)	오름차순 정렬
	= False	내림차순 정렬
inplace	= True	호출한 DF의 정렬 결과를 그대로 적용
	= False(default)	sort_values{ }를 호출한 DF은 그대로 유지, 정렬된 DF를 결과로 반환

```
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Nam
titanic_sorted.head(3)
```

Aggregation 함수 적용

- Aggregation 함수

  : min( ), max( ), sum( ), count( )등

  ⇒ SQL의 aggregation 함수 적용과 유사

  ⇒ 하지만 DataFrame에서는 모든 칼럼에 해당 aggregation을 적용한다는 점이 다름

  - 특정 칼럼에 적용

    : 대상 칼럼들만 추출해 적용

```
titanic_df[['Age', 'Fare']].mean()
```

groupby( ) 적용

- **groupby( )**
  - 입력 파라미터 by에 칼럼을 입력하면 대상 칼럼으로 groupby됨

```
titanic_groupby = titanic_df.groupby(by='Pclass')
```

  - DF에 groupby( )호출하면 DataFrameGroupBy라는 또 다른 형태의 DF 반환

- **SQL group by와의 차이점**
  - DF에 groupby( )호출해 반환된 결과에 aggregation 함수를 호출하면 groupby( ) 대상 칼럼을 제외한 모든 칼럼에 해당 함수를 적용

    ⇒ 특정 칼럼에만 aggregation 함수를 적용하려면 groupby( )로 반환된 DataFrameGroupBy 객체에 해당 칼럼을 필터링한 뒤 함수 적용

```
titanic_groupby = titanic_df.groupby('Pclass')[['Passen
```

  - 서로 다른 aggregation 함수를 적용할 경우에, 여러개의 함수명을 DataFrameGroupBy 객체의

agg( ) 내에 인자로 입력해서 사용

⇒ SQL의 groupby보다 유연성이 떨어짐

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

⇒ agg( ) 내에 입력 값으로 딕셔너리 형태로 aggregation이 적용될 칼럼들과 함수 입력하는 방식

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

✔️ 결손 데이터 처리하기

결손데이터

: 칼럼에 값이 없는, 즉 NULL인 경우 ⇒ 넘파이의 NaN으로 표시

- 머신러닝 알고리즘은 NaN값을 처리하지 않으므로 다른 값으로 대체 해야함
- NaN 값은 함수 연산 시 제외가 됨

isna( )로 결손 데이터 여부 확인

- isna( )는 데이터가 NaN인지 아닌지 알려줌

  ⇒ 모든 칼럼의 값이 NaN이 아닌지를 True/False로 나타냄

  ```
  titanic_df.isna().head(3)
  ```

- sum( )함수를 추가해 결손 데이터 개수를 알 수 있음

  ⇒ True는 1, False는 0으로 변환되어 개수를 구할 수 있음

  ```
  titanic_df.isna().sum()
  ```

fillna( )로 결손 데이터 대체하기

- fillna( )로 결손 데이터를 다른 값으로 대체할 수 있음

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

- 실제 데이터 세트 값을 변경하는 법
  1. fillna( )를 이용해 반환 값을 다시 받기

     eg) titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
  2. inplace = True 파라미터를 추가하기

     eg) titanic_df['Cabin'].fillna('C000', inplace=True)

✔ apply lambda 식으로 데이터 가공

lambda 식

: 함수의 선언과 함수 내의 처리를 한 줄의 식으로 쉽게 변환하는 식

- ':'로 입력 인자와 계산식(반환값)을 분리

```
# 일반 함수
def get_square(a):
  return a**2

# lambda 식
lambda_square = lambda x : x**2
```

- 여러 개의 값을 입력 인자로 사용해야 할 경우에는 보통 map( ) 함수를 결합해서 사용

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

- **판다스 DataFrame의 lamda 식**

  : 파이썬의 Lambda 식을 그대로 적용, 복잡한 데이터 가공이 필요할 경우 사용

```
titanic_df['Name_len'] = titanic_df['Name'].apply(lambda x
titanic_df[['Name', 'Name_len']].head(3)
```

- **if else 절을 사용한 Lambda 식**
  - if 식보다 반환 값을 먼저 기술

    ⇒ lambda x : '반환값' if '조건문' else '반환값'

```
titanic_df['Chld_Adult'] = titanic_df['Age'].apply(lamb
titanic_df[['Age', 'Chld_Adult']].head(8)
```

- else if 이용하기

  : else if는 지원하지 않기 때문에 else 절을 ( )로 내포해 ( ) 내에서 다시 if else 적용

```
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda
titanic_df['Age_cat'].value_counts()
```

- else if가 많이 나와야 하거나 switch case문의 경우

  : 별도의 함수 만들어서 사용

```
# 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 지
# get_category(X)는 입력값으로 'Age' 칼럼 값을 받아서 해당하는
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda
titanic_df[['Age', 'Age_cat']].head()
```